UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

EZGas,
a refuelling assistance application for Android

Author
Carlos Afonso Pérez

Tutor
Prof. Carlos García Rubio

Department of Telematics Engineering

June 2012

Acknowledgements

To Juan and Ana María, coolest parents ever.

To Manu and Baldo, two of the brightest minds I have had the pleasure to meet.

To Carlos, my tutor, for his kindness and patience.

I firmly believe that any man's finest hour, the greatest fulfillment of all that he holds dear, is that moment when he has worked his heart out in a good cause and lies exhausted on the field of battle - victorious.

Vince Lombardi (1913-1970)

Former head coach of the Green Bay Packers

Contents

1 Introduction, motivation and goals **1**
- 1.1 *EZGas*, really? . . . 2
- 1.2 Historical context . . . 2
  - 1.2.1 Rise of fuel prices . . . 2
  - 1.2.2 Popularity of smartphones . . . 2
- 1.3 Structure of this document . . . 3

2 State of the art **5**
- 2.1 Mobile devices . . . 5
  - 2.1.1 IBM Simon . . . 5
  - 2.1.2 Nokia Communicator . . . 6
  - 2.1.3 Symbian . . . 6
  - 2.1.4 BlackBerry . . . 7
  - 2.1.5 iPhone . . . 7
  - 2.1.6 Android . . . 8
- 2.2 Programming languages . . . 11
  - 2.2.1 PHP . . . 11
  - 2.2.2 Java . . . 11
  - 2.2.3 MySQL . . . 12
- 2.3 OpenStreetMap . . . 12
- 2.4 Refuelling assistance applications . . . 13
  - 2.4.1 Feature review . . . 17
  - 2.4.2 Conclusions . . . 17

3 Design aspects **21**
- 3.1 Description of the implemented solution . . . 21
  - 3.1.1 Improved results accuracy . . . 21
  - 3.1.2 Different query methods . . . 22
  - 3.1.3 Thin client-fat server architecture . . . 23
  - 3.1.4 Multiple profiles . . . 23
  - 3.1.5 Schedule filtering . . . 23
  - 3.1.6 Localized interface . . . 23
  - 3.1.7 Clean design . . . 23

3.2 System architecture and design 24
3.2.1 Price data source 24
3.2.2 Result calculation 24
3.2.3 Server design 25
3.2.4 Client design 28
3.3 Use cases 28
3.3.1 Use cases related to vehicle profiles 30
3.3.2 Use cases related to launching requests 30
3.3.3 Use cases related to viewing gas station details 30
3.3.4 Uncategorized use cases 38
3.4 Software requirements specification 38
3.4.1 Functional requirements 38
3.4.2 Non-functional requirements 42

4 Implementation **45**
4.1 Programming languages 45
4.1.1 Server side languages 45
4.1.2 Client side languages 46
4.2 Tools 47
4.2.1 Apache Web Server 47
4.2.2 Subversion 47
4.2.3 Eclipse 48
4.2.4 Android SDK 49
4.2.5 Notepad++ 50
4.2.6 phpMyAdmin 50
4.2.7 osm2po 50
4.3 Server side implementation 51
4.3.1 Database 51
4.3.2 Updater module 51
4.3.3 Web service 54
4.4 Client side implementation 60
4.4.1 Technical services tier 60
4.4.2 Logic tier 61
4.4.3 Presentation tier 64

5 System verification **71**
5.1 Test environment 71
5.2 Test results 72

6 Conclusions and further development **73**
6.1 Final conclusions 73
6.2 Further development 74
6.2.1 Adding other energy sources 74
6.2.2 Calculation of gas stations en-route 74

6.2.3 Providing information about additional services 75
6.2.4 Tracking of statistics 75
6.2.5 Considering traffic status on calculations 76
6.2.6 Augmented reality 76

A Project budget **77**
A.1 Hardware equipment 77
A.2 Software 77
A.3 Services 77
A.4 Human resources 78
A.5 Grand total 78

List of Figures

2.1 The BlackBerry Bold 9650[14] 7
2.2 The iPhone 4S, the most recent iPhone on sale[46] 9
2.3 An screenshot of Ice Cream Sandwich, the latest version of the Android operating system 10
2.4 Screenshots of *Gasofa Barata* 14
2.5 Information overlaid on top of a map in different applications 15
2.6 Tank size is considered by some applications to calculate gross savings after refuelling 16
2.7 Bookmarking gas stations for later review is allowed by some applications 17
2.8 Extra features provided by *Gasolineras España* 18

3.1 A is apparently closer to the user (red dot) than B 22
3.2 The user would need to drive way longer to get to A despite being closer in space 22
3.3 EZGas architecture overview 25
3.4 A common request lifecycle in the original server component design 27
3.5 A common request lifecycle in the final server component design 29
3.6 The server's two-tier logical architecture 29
3.7 Identified use cases in EZGas 30

4.1 Notepad++ 50
4.2 EZGas' server database schema with its only table 51
4.3 The DbGateway class 52
4.4 The grammar describing gas station entries within the CSV file 53
4.5 Class diagram of the updater module 53
4.6 Classes related to a web service request 55
4.7 The Router and RoutingEngineSoapGateway classes 55
4.8 The GasStation and GasStationSchedule classes 56
4.9 An example illustrating the usage of a bounding box to filter results from the database 57

4.10 The ServiceResponse class and the IJsonable interface it implements . . . 58
4.11 An example response from the server . . . 59
4.12 The client's database schema . . . 61
4.13 Persistence layer classes . . . 62
4.14 The client's domain classes . . . 62
4.15 Web serice-related classes within the client application . . . 63
4.16 EZGasMainActivity . . . 65
4.17 EZGasResultListActivity . . . 65
4.18 Detailed information about a specific gas station . . . 66
4.19 Gas stations being shown on a map . . . 67
4.20 Profile management activities in EZGas . . . 68
4.21 EZGasFavoriteGasStationListActivity . . . 68
4.22 EZGasSettingsActivity . . . 69

6.1 Calculation of the best en-route gas stations . . . 75

List of Tables

2.1 Comparison of reviewed existing applications 19

3.1 Geolocation and routing API limits according to different providers 26
3.2 UC001 - *Creating a vehicle profile* 31
3.3 UC002 - *Editing a vehicle profile* 31
3.4 UC003 - *Deleting a vehicle profile* 32
3.5 UC004 - *Setting a vehicle profile as active* 32
3.6 UC005 - *Performing a request* 33
3.7 UC006 - *Requesting the closest gas stations around the user* . 34
3.8 UC007 - *Requesting the best gas stations around the user given a fixed cost* 35
3.9 UC008 - *Requesting the cheapest gas stations around the user for a full refill* 36
3.10 UC009 - *Viewing details of a gas station* 37
3.11 UC010 - *Bookmarking a gas station* 37
3.12 UC011 - *Removing a bookmarked a gas station* 38
3.13 UC012 - *Navigating a map* 39

4.1 Expected parameters by the web service 60

A.1 Price and cost of hardware equipment 77
A.2 Price and cost of services 78
A.3 Price and cost of human resources 78
A.4 Total cost of the project 78

Chapter 1

Introduction, motivation and goals

The ultimate goal of this project is to build *EZGas*, an application which helps users to determine which gas station is the most suitable for refuelling their vehicles, given different criteria such as the user's position or the total cost of the operation. The application will be targeted at handheld devices running the Android operating system and will communicate with a web server to calculate the best choices for each request.

The system will therefore be composed of two main components, the details of which are described in chapter 4:

- the Android application, in charge of capturing and collecting user's relevant information, building and sending requests, processing replies and presenting the user with the resulting options and some additional features.
- the web server, responsible of listening to and processing incoming requests, determining and sorting the best matches according to the user's defined criteria and delivering the results. The web server will also be in charge of keeping an up-to-date list of gas stations and the prices of each type of available fuel.

The author feels that such a tool can be of great interest and utility for a great amount of users nowadays as fuel prices are breaking historic maximums, the purchasing power of most citizens has sharply decreased due to the current global financial crisis and, in parallel, the number of smartphone owners increases every day.

Furthermore, this is a great opportunity to enhance and contribute to the current set of available applications on the market which aim to fulfill the same task.

This brief chapters draws some context for the project and presents the structure of this document.

1.1 *EZGas*, really?

The project has been given the name of *EZGas* because, well, it is all about gas; not *gas* as in *one of the three states of matter* but as in *gasoline*, as colloquially referred to by most people living in North America.

The "EZ" portion is a play on how the word *easy* is pronounced in English, because EZGas is supposed to facilitate the task of finding where to refuel the user's vehicle driving the fewest distance or spending the least amount of money.

1.2 Historical context

A more detailed study on the current situation of different concepts that directly affect this project is presented in the next chapter. The following paragraphs are meant to give the reader a quick perspective on why EZGas was conceived and why its development was deemed appropriate by the author.

1.2.1 Rise of fuel prices

Back in 1990, a barrel of Brent crude (42 US gallons, or almost 159 liters), one of the reference crude oil benchmarks, averaged a cost of $24. Twelve years later, the same barrel costs slightly under $119. That is an increase of about 500%.

Gasoline and diesel fuel are two derivatives of oil crude which are mostly used to power road vehicles. The combined worldwide consumption of these two types of fuel in 2008 added up to 46 millions of barrels per day.

Needless to say, retail prices of gasoline and diesel fuel are directly related to that of crude oil. For instance, the average price, without considering taxes, of unleaded gasoline in Spain increased 121% just between 2010 and 2011. Diesel fuel behaved similarly with an increase of 126% during the same time span.

On the other hand, in 2011 there were well over 20 millions of vehicles driving on Spanish roads and highways. This is an indicator of how dependent Spanish citizens are on their private vehicles to move around town, and also of how expensive is the substance that powers them.

1.2.2 Popularity of smartphones

During the last five years the mobile phone industry has witnessed an impressive development of some devices, known as *smartphones*, which feature the processing power of not-so-old computers, huge multitouch screens and which fit right into users' pockets. These devices have opened new market opportunities as they have the power to run regular applications such as web

browsers, email clients or text processors but also have an added component of mobility.

Nowadays developers can make use of the features of these devices to create an enhanced user experience, such as the location provided by a GPS antenna or the contents of a user's music library.

1.3 Structure of this document

The document is divided into different chapters, each of them being dedicated to a specific part of the development process.

Chapter 2 presents a detailed study about the current context surrounding the project, such as the evolution of fuel prices, cell phones and smartphones and geographical information platforms. Furthermore, since there is already some software which serves a very similar purpose as EZGas, a comparison of some existent applications is carried out which presents their features as well as their strong points and drawbacks.

Chapter 3 is dedicated to the high-level design phase of the project. First a quick specification overview will be presented, only to be followed by use cases and requirements which will drive the overall software implementation.

Chapter 4 dives into the implementation itself. It describes the architecture of the EZGas project, enumerates the development tools and provides detailed insight on how each feature of EZGas works behind the scenes.

Chapter 5 briefly explains the validation and testing process of the application.

Finally, chapter 6 contains final thoughts from the author regarding the project, as well as some ideas which could be implemented in the future.

In addition, annex A presents an estimation of the project costs in the shape of a budget.

Chapter 2

State of the art

This chapter will describe the different technologies and concepts which affect or are related in some way to this project.

2.1 Mobile devices

Once upon a time there was a cell phone manufactured by Nokia which went by the model name of 3310. It featured a calculator, a stop watch, SMS support and an awesome game called *Snake*. If that was not enough, users could customize their phones by changing the monochrome screensavers and downloading custom ringtones[33]. This feature-packed, techonologically-advanced device sold massively well worldwide and became one of the most popular cell phones ever marketed[35].

Incredibly enough this happened not so long ago, as the 3310 was first launched at the end of 2000. In the past 12 years there has been a huge revolution within the mobile phone industry which has translated into what today people know as *smartphones*. These evolved phones allow users to browse websites, check emails, watch high-definition videos, participate in videoconferences and play games against other opponents who might be anywhere in the world. And all this power is concentrated into a device which fits in a person's hand.

But well before Apple released the first version of their hugely popular iPhone, other smartphones appeared in the market with varying success. This section will attempt to walk through their history and mention some of the most influential devices that were built.

2.1.1 IBM Simon

IBM and BellSouth joined efforts back in 1993 to sell the Simon, a device which attempted to merge functionality from a regular mobile phone, a PDA and a pager. It featured a touchscreen instead of buttons and a stylus which

was used to control all its applications, among which there were a notepad, a calculator or a calendar.

Despite it being huge and heavy, looking like a brick, and having a prohibitive cost (starting at $899 but costing almost $1100 if it was bought in an area which was not served by BellSouth[36]), the Simon is considered the first smartphone ever made.

2.1.2 Nokia Communicator

The next milestone in the history of smartphones came in 1996 in the shape of the Nokia 9000, which was the first model of the Communicator series[34]. The 9000, just like the rest of the Communicator devices, features a top lid which contains a regular display and an standard phone keyboard. Additionaly, this lid can be flipped over and, when doing so, a high-resolution display and a full QWERTY keyboard are revealed.

Both the 9000 and its successors 9110 and 9110i ran on the GEOS operating system. This changed with the arrival of Symbian.

2.1.3 Symbian

Symbian was an open operating system which was seen first running on the Ericsson R380 back in 2000, even though it became popular worldwide as the operating system of a number of Nokia devices, such as the rest of the Communicator line starting from the 9210[38].

Symbian was derived from an operating system called EPOC which was first developed by a company named Psion between the 1980s and the 1990s. EPOC was used in the company's devices and PDAs and in 1998, just as the OS was in its sixth release, the software branch of Psion was spun-off and renamed as Symbian as a joint effort between Psion, Nokia, Ericsson and Motorola, even though Psion parted from the project in 2004.

The following years witnessed the birth of different graphical user interface platforms such as Series 60, 70 and 80 by Nokia or UIQ by UIQ Technology. Newer releases of the Symbian OS also brought significant enhancements and new features such as OpenGL ES, mandatory application code signing, IPv6, WiFi and SQLite databases, among others.

However, in late 2008 Nokia acquired the company behind the development of Symbian, Symbian Ltd, and became a key contributor to its development. It was then when the Symbian Foundation was created in an effort to release Symbian's source code to the public. Symbian therefore became not only an operating system but a full mobile platform, as it merged the core Symbian OS and Nokia's S60 user interface platform.

In February 2011 Nokia partnered with Microsoft in order to adopt the Windows Phone 7 OS in it smartphones which, ultimately, will result in the abandonment by Nokia of the Symbian platform[40]. A few months later

Nokia announced that it would outsource the development of Symbian to Accenture up to 2016[17].

2.1.4 BlackBerry

The BlackBerry is a series of smartphones and tablet devices created by Canadian company Research In Motion (RIM). The first BlackBerry ever built was first introduced in 1999 and basically consisted of a wireless email client with the size of a pager device[12].

RIM initially focused on the *push* email functionality oriented to the business consumer, but as of today all BlackBerry models feature the typical smartphone capabilities: Internet browsing, gaming, multimedia playing and other functions which are characteristic of any PDA.

Figure 2.1: The BlackBerry Bold 9650[14]

BlackBerries run on the proprietary operating system BlackBerry OS, also developed by RIM. The current stable release is 7.1 and is used by the latest Bold and Torch models (9900 and 9850 respectively). Develpers can write custom applications for BlackBerry devices and publish them through the BlackBerry App World service. The newest and upcoming version of the OS, BlackBerry 10, supports applications based in HTML5[21].

By April 2012, BlackBerry OS was taking a 6.1% of the mobile operating system market share[43].

2.1.5 iPhone

The iPhone is Apple's take on smartphones and is considered to be one of the most influential devices to the mobile device industry ever created. It is not the first mobile device produced by the company, though, as the Newton line of devices was launched in 1993.

In the years leading up to the release of the iPhone, Apple had achieved massive worldwide success with the iPod, a portable media player, and iTunes, a software for synchronizing the contents of computers and iPods and also for purchasing and downloading new material from the iTunes Store. Apple joined Motorola in an effort to translate such business model into the mobile phone industry, and the ROKR E1 was born.

The ROKR, however, resulted to be a huge flop and was criticized because, despite giving the user the ability to upgrade its stock 512 MB memory card, the device was firmware-limited to a maximum of one hundred songs[41]. The ROKR was also hit by critics due to its low transfer speed.

Due to this commercial failure Apple discontinued support for the ROKR and decided to create a mobile phone of their own, a development codenamed *Project Purple 2*[4]. This decision, alongside previous work on touchscreens by Apple engineers, resulted in Steve Jobs' announcement of the first iPhone during a Macworld convention in 2007[44].

The original iPhone featured a 9cm capacitive multi-touch screen (with a resolution of 320x480 at 163 ppi) and a was characteristic for having only one phyisical button (apart from the wake-up and volume control buttons) which was named the *Home button*. It also included a 2-megapixel camera.

Newer generations of the iPhone have appeared since then, namely the iPhone 3G, the iPhone 3GS, the iPhone 4 and the iPhone 4S (which apperas in figure 2.2, each of them bringing a feature overhaul in respect to their antecessors. For example, the iPhone 3G incorporated a GPS antenna and support for 3G data transfer, the iPhone 3GS included video recording support, the iPhone 4 featured two cameras (one of them having 5 megapixels) and a gyroscope and the iPhone 4S presented the Siri voice recognition system.

All of them run on the iOS, Apple's mobile operating system which derives from a core Mac OS X component known as Darwin [20]. iOS-powered devices, which also include newer iPhones and the iPad, represent a 23.85% of the global OS market share[43].

Third-party applications are available for downloading and/or purchasing at the App Store. Latest reports indicate that, as of March 2012, the App Store featured 550,000 *apps* apps and grossed over 25 billion downloads since its opening in July 2008[32].

2.1.6 Android

Android is an open-source operating system originally developed by Android, Inc. from 2003. Google purchased Android in 2005 and became the main developer of the project until today[19]. Google also was one of the cofounders of the Open Handset Alliance, a consortium of companies created in 2007 which aims to develop open standards for mobile devices[2]. Android's code is released to the general public by Google under the Apache

Figure 2.2: The iPhone 4S, the most recent iPhone on sale[46]

license.

The Android operating system features an adapted Linux-based kernel written in C and uses a Dalvik virtual machine. Code running on the Dalvik VM (contained in *dex* files) is translated from Java bytecodes, meaning that applications are usually written in Java. This is not always the case as Android also features a toolkit for incorporating code in C and C++ into an application[19], called the Native Development Kit (NDK).

Developers make use of the Android SDK to build their own applications, and users can install them by either acquiring them at Google Play (previously known as the Android Market) or by downloading the .apk file from a third-party provider[19]. As of May 2012, Google Play was reported to contain over 500,000 applications in its catalogue and had served over 15 billion downloads[30].

Many mobile devices are powered by Android, not only smartphones but also tablets, netbooks, ebooks and TV platforms. The first marketed device to do so was the HTC Dream back in 2008[19].

Several versions of Android have appeared since then, each of them being named after a different dessert:

- 1.5 *Cupcake*, released in April 2009
- 1.6 *Donut*, in September 2009

- 2.0 *Éclair*, in October 2009, and also version 2.1, in January 2010
- 2.2.0 (and above) *Froyo*, in May 2010
- 2.3 (and above) *Gingerbread*, in December 2010
- 3 (and above) *Honeycomb*, in February 2011 (oriented to tablet devices only)
- 4 (and above) *Ice Cream Sandwich*, in October 2011 (shown in figure 2.3

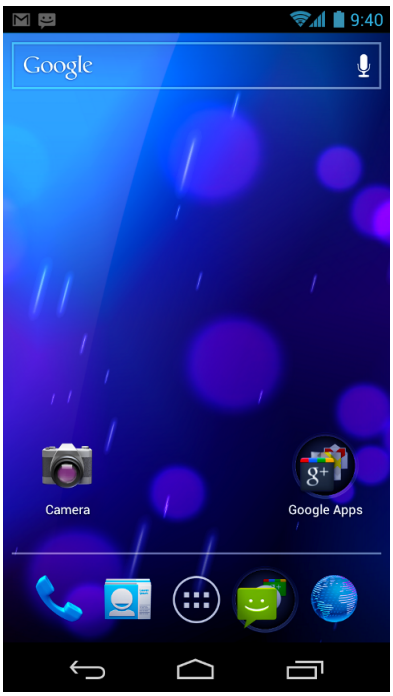


Figure 2.3: An screenshot of Ice Cream Sandwich, the latest version of the Android operating system

Google has marketed some devices oriented to Android developers. These devices are unlocked, come with stock system images and free of carrier modifications, feature an unlocked bootloader (allowing users to flash any system image they wish) and are usually released alongside a new version of Android. These devices are the Android Dev Phones 1 (Android 1.0) and 2 (Donut), the Nexus One (Froyo), the Nexus S (Gingerbread) and the Galaxy Nexus (Ice Cream Sandwich).

As of March 2012, reports claim that Android represents a 23.79% of the global mobile OS market share[43] with over 330 million devices running it[3].

2.2 Programming languages

Since the advent of the first computer thousands of programming languages have been developed. Of course, their usage and areas of application differs from language to language: only a few of them are widely used today (C, Java) whereas others are almost extinct (B); some of them are aimed at systems programming (C, C++), whereas others are used for logic (Prolog) or artificial intelligence programming (Lisp). There are even esoteric languages which were created for the purposes of experimenting or just for fun, such as FALSE or brainfuck.

The age of a programming language does not necessarily influence whether or not its usage is decaying or at the crest of the wave: Fortran appeared back in 1954 and it is still widely used today (either the original or one of its dialects), whereas Pascal has been suffering a significant usage drop-off even though it was created in 1970.

This section presents some detailed information regarding the programming languages in which EZGas has been implemented.

2.2.1 PHP

PHP, which stands for *PHP: Hypertext Preprocessor*, is a general-purpose scripting language created in 1995 by Danish programmer Rasmus Lerdorf. It is an alternative to JSP and ASP, and was originally aimed to be run on servers and to produce Web pages dynamically. In such scenarios, PHP code is embedded into HTML markup and is later interpreted by the PHP processor. Finally, pure HTML content is generated and sent to the client[18].

This is the most common usage of PHP. However, it is also used for command-line scripting, without the need for a server, and for building desktop applications, even though there are better alternatives for the latter.

The strong points of PHP are its portability, as it runs on all major operating systems, and the huge number of available libraries which allow for image processing, parsing and writing XML files, connecting to different database providers or processing text.

PHP is free, open source and ranked 6th at the TIOBE ranking on May 2012[42]. It is usually the P in the LAMP software stack (other times being Perl or Python).

2.2.2 Java

In early 1991, a team of Sun Microsystems engineers, among which James Gosling, Mike Sheridan and Patrick Naughton were present, set out to develop a system which could control intelligent electronic consumer devices, and more specifically, set-top boxes. Development efforts were codenamed Stealth Project at first, to be renamed later as the Green Project[1].

James Gosling planned to use C++ as the programming language of the Green Project. However, he later deemed it to be inadequate and decided to write an independent language which was originally named Oak. Since Oak was to be run in a number of different devices it had to be platform-independent, so instead of designing it as a compiled language, Oak would be interpreted and converted to a middle-layer format known as *bytecode*.

The Green Project, despite being a technological achievement, failed to be commercially successful several times up to 1994, when Sun executives decided that the project was too advanced for the interactive TV market of the time, which was considered immature.

Some of the project developers decided to give Oak (now named Java due to coypright issues) another chance, this time as a programming language to be used in web browsers. This new orientation was well received after its release in 1995 and Java quickly became popular.

From a technical point of view, Java is an object-oriented, distributed, multi-threaded programming language which is part of the homonymous platform[7]. Java programs do not directly run on the operating system. Instead, Java code is compiled into *bytecodes* (a sort of intermediate-level code) and run on a Java Virtual Machine (JVM). This effectively makes Java a platform-independent language since Java code can run on any operating system for which the JVM is available.

According to Oracle, the current owner of Java after purchasing Sun Microsystems, the Java Runtime Environment is downloaded 930 million times each year[8].

2.2.3 MySQL

MySQL is one of the world's most popular relational database management systems (RDBMS). First released in 1995 and written in C/C++, it supports multithreading, provides both transactional and nontransactional engines, features APIs for a broad set of programming languages, uses a subset of ANSI SQL 99 and is able to run in different operating systems among which Linux, Mac and Windows can be found[11].

MySQL is known to be used in highly popular websites such as Wikipedia, YouTube, Facebook and Twitter among others[9].

2.3 OpenStreetMap

OpenStreetMap, or OSM, is a collaborative database of geographical information. Anyone from professional cartographers to enthusiasts contribute to the project by uploading geographical data, which may be collected from different sources such as GPS tracks or aerial photography. This information can be later used in a wide range of applications, the most common ones being map rendering and routing.

The OSM project was first launched back in 2004 as a response to the restrictions of use imposed on most of the available geographical data at the time. It was inspired by platforms such as Wikipedia, which is one of the best known examples of collaboratively edited websites.

Data in OSM is organized in four different primitive types:

- **nodes**, which are single points on the Earth's surface defined by a latitude and a longitude. A node can represent anything from a road junction, a point of a park's boundaries, a bus stop or a power line post.
- **ways**, a sequence of nodes representing both linear and polygonal features, such a street, a railroad section, a primary school or a block of buildings.
- **relations**, which group together a set of nodes, ways and even other relations which share some features. For instance, a long highway which spans several ways can be defined using a relation.
- **tags**, which are key-value pairs used to add further information about nodes, ways and relations. Such a tag could be `maxspeed=50`, which is usually applied to nodes being part of highways or railways and indicates the speed limit at the given node.

OSM's backend is powered by a PostgreSQL database which includes the PostGIS extension, and data is added and edited using an interface to a Ruby on Rails web application. There is a great number of third-party software developed around OSM

2.4 Refuelling assistance applications

EZGas is not a groundbreaking concept: when the project started, there were already a few applications available which calculated the best gas station where users could refill their vehicles' gas tanks; furthermore, some new ones have appeared over the course of the last months, whereas some existing ones have improved in some way. However, EZGas aims to enhance some of the functionality already provided by these alternatives as described in chapter 3. This section presents some of the existing applications available for the Android platform and which are oriented to the Spanish market.

All applications under study present a basic set of common features:

- they present price data for several different types of fuel, and
- they search for gas stations close to the user's position by means of the phone's GPS, cell positioning or other methods

However, only some of them extend this functionality in specific ways.

The most basic live example of such an application can be found in *Gasofa Barata*, by Insaga. *Gasofa Barata* presents price data for gas stations around the user in a simple way. Data usually includes the gas station name, price for all gas types and the distance from the user's known position, even though some customizations can be made. Search radius is limited to a predefined set of 2.5, 5 and 10 km. Some screenshots of *Gasofa Barata* are shown in figure 2.4.

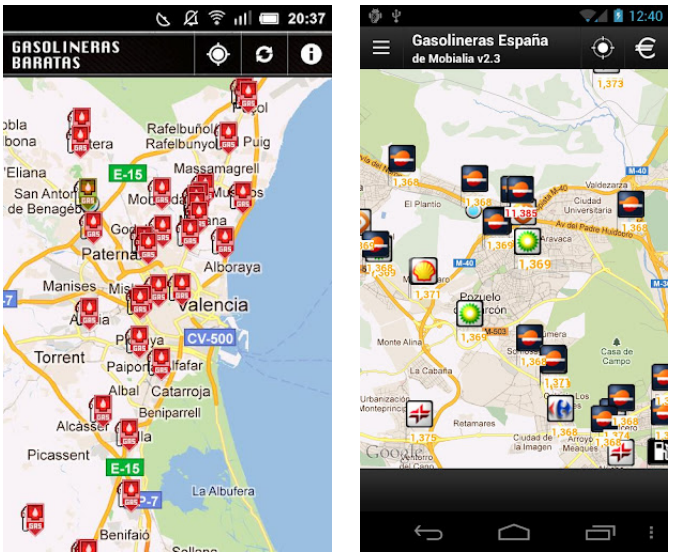
(a) Location selection screen (b) Result list

Figure 2.4: Screenshots of *Gasofa Barata*

The rest of the applications under analysis are considerably more ellaborated. All of them present information as overlay graphics on a map, as shown in figure 2.4, although *Gasolineras España*, by Mobialia, is the only one which displays the gas station logo on the map for a quick identification.

As mentioned earlier, all of them allow to query prices for different types of fuel; however, how this feature is implemented greatly differs from one application to another. For example, *Gasolineras Baratas*, by Mobilendo, and *Gasolineras España* allow to set the preferred fuel type as an application preference, whereas *Lo más Barato: Gasolina*, by Waiyaki Studio, and *Gasolina Barata*, by Amidasoft, allow the user to choose at the time of the request.

The calculations performed by all applications seem to consider the price per liter, that is, the cheapest gas station is the one with the lowest price, even though *Gasolineras Baratas* and *Gasolineras España* also consider the size of the vehicle's fuel tank to calculate the total cost of the refuelling and obtain the savings of the operation. Such information can be seen in figure 2.4.

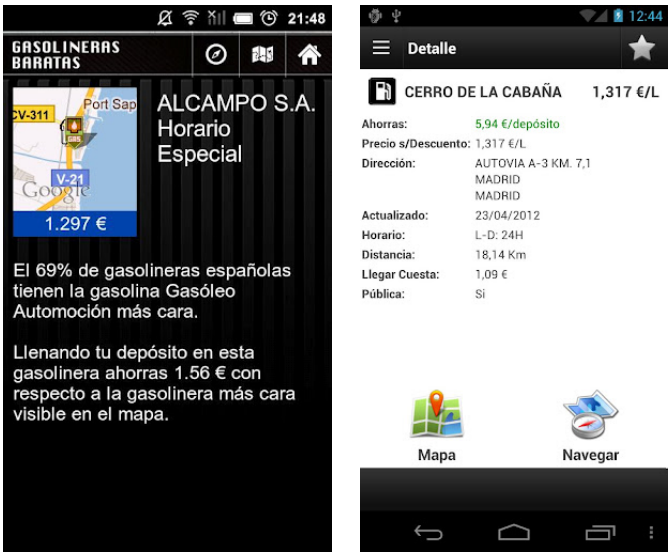


(a) *Gasolineras Baratas* (b) *Gasolineras España*

(c) *Gasolina Barata* (d) *Lo más Barato: Gasolina*

Figure 2.5: Information overlaid on top of a map in different applications

Figure 2.4 also shows how *Gasolineras España* does consider the cost of getting to the gas station from the user's current position, even though this cost is not used to determine the most economic option.



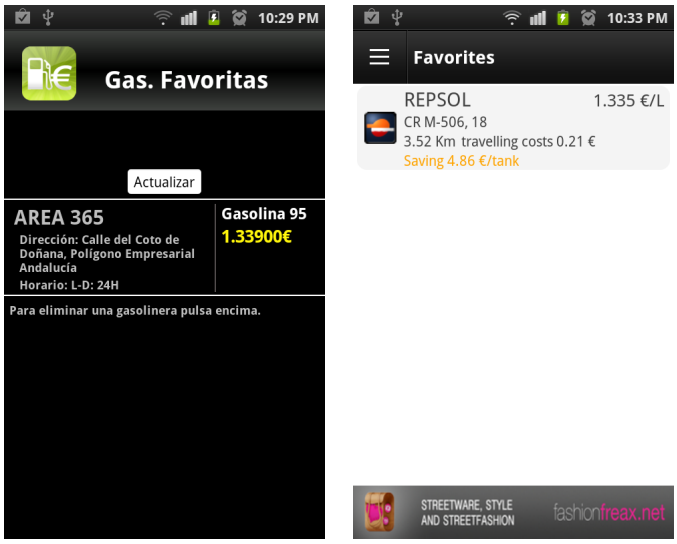
(a) *Gasolineras Baratas* (b) *Gasolineras España*

Figure 2.6: Tank size is considered by some applications to calculate gross savings after refuelling

Moving on to the additional features, *Gasolina Barata* presents an interesting functionality which displays all gas stations which are close to a user-defined route between two points, even though no special distinction is made between them nor a "best option" for the user is determined. This is a feature which could be implemented in EZGas in future versions, as discussed in chapter 6.

There are three applications which allow the user to bookmark a specific gas station for later review. These applications are *Gasolineras España*, *Gasolineras Baratas* and *Gasolina Barata*. Such a functionality can be seen working in figure 2.4.

Gasolineras España is the application which offers the greatest number of extras. For example, results are tagged with either green, yellow or red text depending upon the savings per tank refill. *Gasolineras España* also presents the user the choice to hide or show outdated prices and to show some brands of fuel while hiding others. All these features can be seen in figure 2.4.



(a) *Gasolina Barata* (b) *Gasolineras España*

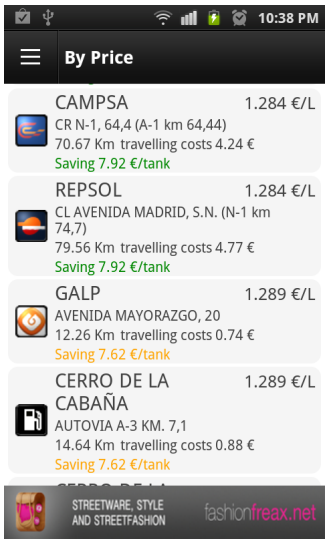
Figure 2.7: Bookmarking gas stations for later review is allowed by some applications

2.4.1 Feature review

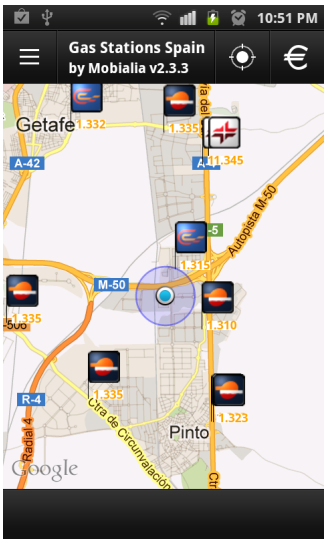
A final summary of the features provided by the reviewed applications can be found in table 2.1.

2.4.2 Conclusions

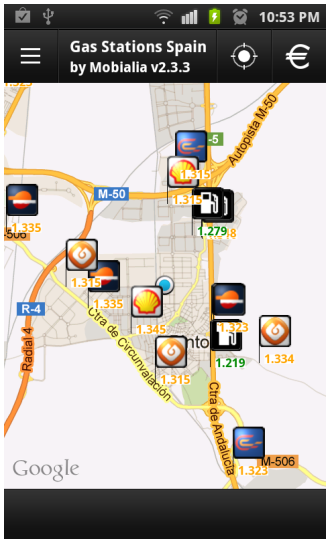
All the information detailed in this section shows the different approaches that developers have taken when developing their applications. Even though some good ideas have already been developed, there is definitely some room for improvement. Some of the described functionality has been used as reference when designing EZGas, whereas some has been discarded due to time constraints and some features have been enhanced or added. The details of the features implemented in EZGas are described in chapter 3.



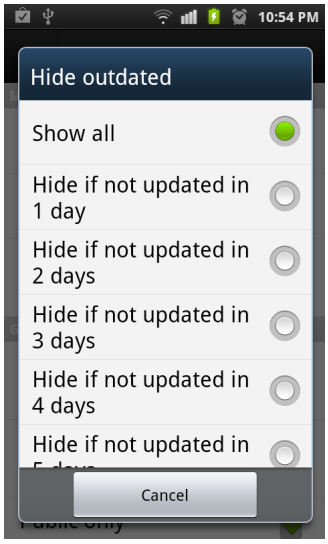
(a) Color tagging by savings



(b) Displaying only Campsa, Cepsa and Repsol gas stations



(c) Displaying all gas stations



(d) Different options to filter outdated prices

Figure 2.8: Extra features provided by *Gasolineras España*

	Gasofa Barata	Gasolineras España	Gasolineras Baratas	Gasolina Barata	Lo más Barato: Gasolina
App version	3.3	2.3.4	1.3.0	2.2.0	1.5.1
Platform	Android	Android	Android	Android	Android
Minimum platform version	1.5	1.6	2.2	2.2	2.1
Price	Free	Free	Free	€0.59	Free
Gas types	6	6	5	4	8
Considers trip cost	No	Yes	No	No	No
Bookmarks	No	Yes	Yes	Yes	No
Map overlays	No	Yes	Yes	Yes	Yes
Navigation	No	Google	Google	No	Google
En-route gas stations	No	No	No	Yes	No
Additional features	None	Result coloring, outdated gas stations filtering, brand filtering, user discounts	Saving tips	None	Saving tips

Table 2.1: Comparison of reviewed existing applications

Chapter 3

Design aspects

3.1 Description of the implemented solution

EZGas is an application which runs on Android devices and which helps users to determine where to refuel their vehicles according to different criteria. This section presents the implemented functionality in basic terms.

3.1.1 Improved results accuracy

The main improvement of EZGas over its existing alternatives is how it calculates the most adequate gas stations for a given request. As described in chapter 2, other applications only consider fuel prices when determining the most economical options; EZGas not only takes that into account, but also the distance between the user and each gas station and what cost does travelling that distance imply.

EZGas also provides enhancements related to distance calculations. The rest of applications, when calculating the closest gas stations to the user, use a linear distance between both points, whereas EZGas always considers the real driving distance as long as it is available. The rationale behind this decision is simple: accuracy; the user might be very close to a gas station but due to the road network he/she might need to travel a longer distance to get there.

This fact is graphically described in figure 3.1, where a possible scenario is described. The user is at the red mark and is running out of fuel, so he/she wants to know where is the closest gas station. Other applications will tell the user to head towards A as it is linearly closer than B. However, figure 3.2 shows how the real distance to A is greater than that to B. B is therefore a better option, but the rest of applications won't consider this and the user will run out of fuel on his way there, with all the associated distress and drama. EZGas will, on the other hand, advise the user to head towards B, where he/she will be able to refuel the car and move on as if nothing had happened.

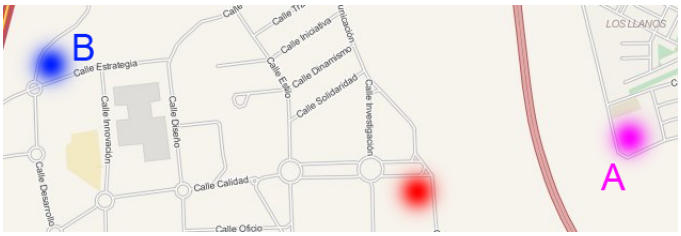


Figure 3.1: A is apparently closer to the user (red dot) than B

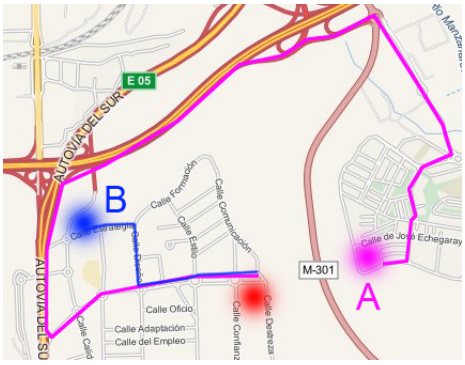


Figure 3.2: The user would need to drive way longer to get to A despite being closer in space

The author considers that the features that have just been described help to deliver more accurate results to the user, which is the ultimate goal of an application of this kind.

3.1.2 Different query methods

Users can discover which are the best gas stations around them in different ways.

Closest gas station

The simplest method: the closer the gas station is to the user, the better.

Fixed cost

This option is suited for people who use to spend a fixed amount of money every time they stop at a gas station. With this modality users specify how much they are willing to spend and EZGas calculates the gas stations where the most volume of fuel can be obtained, considering as well the amount of fuel consumed on the trip.

Filling the tank up to the top

This option is better suited for users who want to fill up their vehicles' tanks. In order to provide the best matches, EZGas asks the users how much fuel there is left in their tanks and calculates which gas stations are the cheapest considering the trip cost and the refuelling cost.

3.1.3 Thin client-fat server architecture

Battery consumption and network usage are two paramount attributes to consider when developing for mobile devices. EZGas follows a client-server architecture where the client asks just for what it needs and the server manages all the heavy lifting. This means that no big files are downloaded to nor any CPU-intensive calculation (such as pathfinding) is carried out on the device itself, helping to preserve battery life and to save data bandwidth.

3.1.4 Multiple profiles

EZGas allows users to define *vehicle profiles*. Each profile determines the fuel type of a given vehicle, as well as its consumption rate and the size of its fuel tank. Of all the defined profiles, only one is considered to be active and is used as a reference when asking the server for results. Since the active profile can be changed at any time, this permits users to set different configurations associated to different vehicles only once and switch between them with a couple of finger taps.

3.1.5 Schedule filtering

EZGas lets the user decide whether or not the server should include closed gas stations in its calculations, as long as their schedules are known. This helps the user in the event that he/she plans to refuel immediately, as driving to a closed gas station represents a waste of time and money.

3.1.6 Localized interface

EZGas automatically detects the language of the underlying operating system and presents its user interface either in Spanish (when the device uses Spanish) or in English (in any other case, including, of course, English).

3.1.7 Clean design

The user interface has been designed to provide a simple user experience, following Android design guidelines and using common themes, styles and colors in every layout.

3.2 System architecture and design

As discussed previously, EZGas has been designed following a thin client-fat server model. The rationale behind this decision is basically to avoid as many calculations to be performed on the device as possible, as well as to keep data usage to a minimum. An schematic overview of such architecture is presented in figure 3.3

Accomplishing the previous statement might be hard if no server was present, mainly due to two reasons.

3.2.1 Price data source

Fuel prices are freely provided by the Spanish Ministry of Industry, Tourism and Commerce (MITYC) as a series of compressed files, one per each fuel type, which contain the prices of all gas stations in the country and which on average have a size of 150 KB. If a user were to check the price of a specific gas station the whole file would need to be downloaded. Furthermore, if the same user wished to check prices of other fuel types, their corresponding files should be downloaded as well. Since prices are updated almost daily (and having the latest price information is crucial), this operation would repeat every day.

Although price files are compressed, they come in a custom CSV-based format which would need to be parsed and converted in order to efficiently calculate the best options for the user. This comes at the cost of an increased CPU usage, which affects battery lifetime. Carry this facts into a daily, continuous usage of the application and EZGas could end up using about 9 MB of data per month and executing intensive decompressing-parsing operations directly on the device.

If a server was used, the mobile application would only need to ask the server for what it needs and wait for its reply. This would reduce the bandwidth usage to a few hundreths of bytes per request, so if the average user performed 10 requests per day and each request consumed 150 B of bandwidth, the total monthly cost would be under 50 KB (which is only a 0.05% of the no-server version).

3.2.2 Result calculation

What calculations EZGas performs to get the best choices is discussed in chapter 4, but for the purposes of this section it is enough to realize that they are somewhat CPU-intensive as they deal with path-finding operations among others.

Having the phone application carrying out those calculations by itself would negatively impact on the battery usage, whereas by using a server

instead the same application would only need to provide some parameters and wait for the already-baked reply.

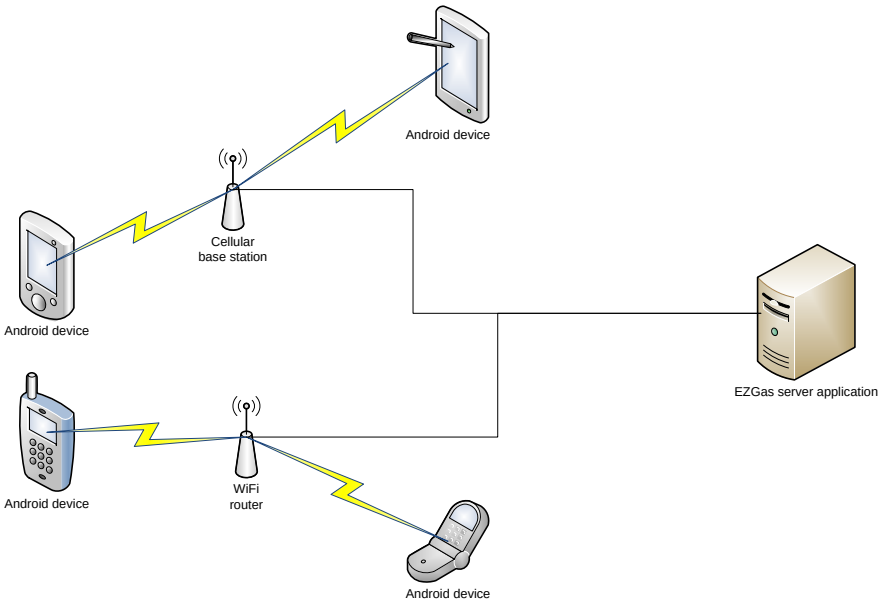


Figure 3.3: EZGas architecture overview

3.2.3 Server design

The server itself is composed of different software elements which work together to deliver the required results. The initial design considered the following components:

- the service endpoint, where all requests are addressed,
- the request processor, which performs all the actual calculations,
- the security manager, which is in charge of authenticating requests and denying unauthorized use of the service,
- the storage facility (mainly a MySQL database, as discussed in chapter 4),
- the price updater, which perdiodically grabs the available prices from the government source (MITYC) and stores the new values into the storage facility, and
- the geolocation API gateway

Provider	Daily limit	Notes
Google Directions	2,500[26]	None
Bing Maps	50,000[6]	500,000 req./year[6]
Yahoo! Maps	Discontinued	None
MapQuest Open Directions	No limit[28] [1]	None

Table 3.1: Geolocation and routing API limits according to different providers

Geolocation APIs

It is worth dedicating some time to explain the last enumerated component, the geolocation API gateway. This component was included at first because, originally, in order to fullfil the real-driving-distance requirement (as described in section 3.1) some free third-party routing APIs were considered. These APIs included the likes of Google, Yahoo! and Bing Maps.

The original idea was to contact one of the servers every time a distance was required by EZGas instead of having a local component commited to such a task, and the gateway component would be in charge of unifying the responses of all APIs into a single format and passing them on to the processor. EZGas was to use more than one API provider mainly for two reasons:

- every API imposed a daily request limit (the details of which, as June 2012, are presented in table 3.1); this way the server could switch between API providers once the daily cap was hit on one of them, and
- to provide a failure-safe mechanism, as any of the providers could unexpectedly go down at any time without prior notice

By using the aforementioned APIs, any request would present the following lifecycle within EZGas' own server, which is graphically described in figure 3.4:

1. the request would reach the web service endpoint,
2. the request would be authenticated by the security manager,
3. if authenticated, it would be passed on to the processor, which would extract the parameters and determine what is being asked for,
4. the processor would make use of the database in order to fetch information about gas stations; it would also request driving distances through the use of the API gateway,
5. finally, once the calculations are done, the processor would construct a response and return it back to the client

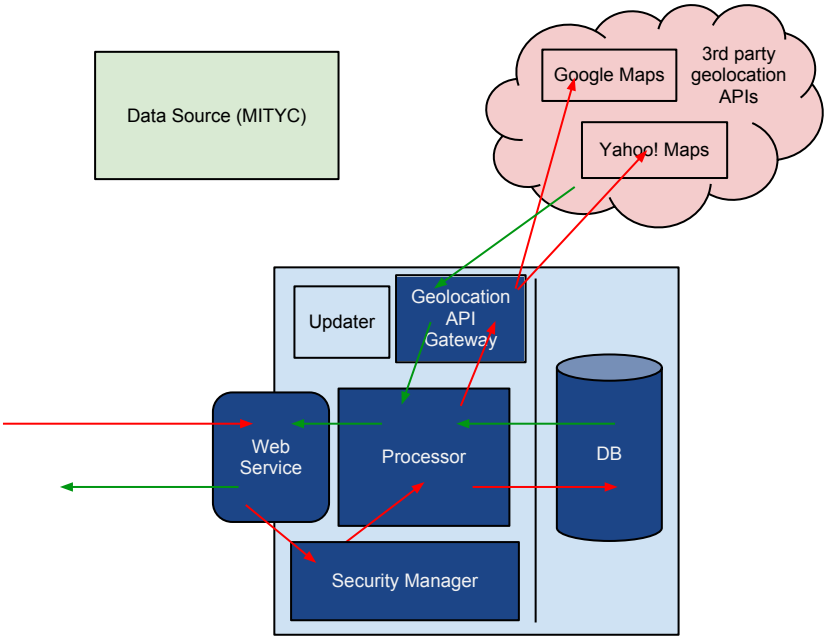


Figure 3.4: A common request lifecycle in the original server component design

However, the final version of EZGas does not make use of said APIs because of license restrictions and how they interfere in the way their data was to be used. For example, the Google Directions API Terms of Service states the following, under section 10.1.1.g[25]:

> You must not use or display the Content without a corresponding Google map, unless you are explicitly permitted to do so in the Maps APIs Documentation, or through written permission from Google. In any event, you must not use or display the Content on or in conjunction with a non-Google map. For example, you must not use geocodes obtained through the Service in conjunction with a non-Google map. As another example, you must not display Street View imagery alongside a non-Google map, but you may display Street View imagery without a corresponding Google map because the Maps APIs Documentation explicitly permits you to do so.

It is clear at this point that the server would not be using the distances provided by Google alongside a map, let alone a Google map. These usage conditions apply in a more-or-less restrictive way in every other considered API.

Due to these restrictive terms of use, and in order to avoid any potential legal issues, another solution had to be found. This is why the final server architecture incorporates its very own router engine (the *osm2po* tool, described in section 4.2) which makes usage of the free-to-use OpenStreetMap data set. The final request lifecycle and server component configuration is displayed in figure 3.5.

Functional architecture

From a logical point of view, the server can be organized as a two-tier architecture in which there is a logic layer and a technical services (or persistence) layer, as described in figure 3.6.

3.2.4 Client design

As for what concerns the client side there are no specific architectural considerations due to the simple nature of the application. However, detailed information on how it is organized internally can be found in chapter 4.

3.3 Use cases

This section presents all the use cases identified in EZGas, the full set of which is presented in figure 3.7. A single actor has been determined to

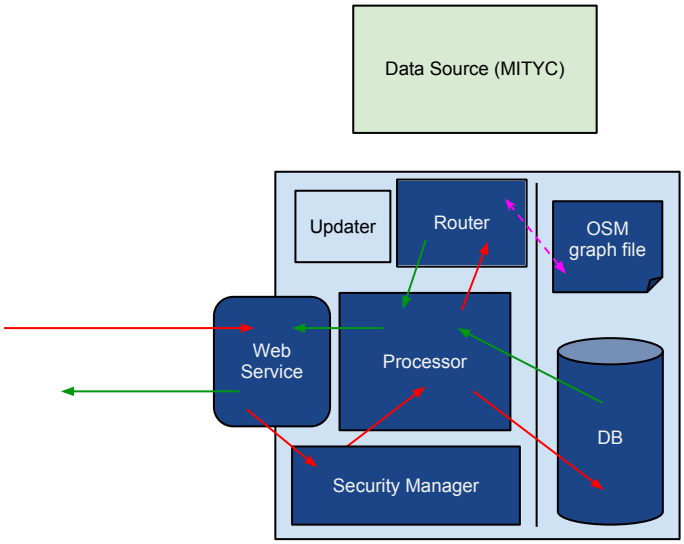


Figure 3.5: A common request lifecycle in the final server component design

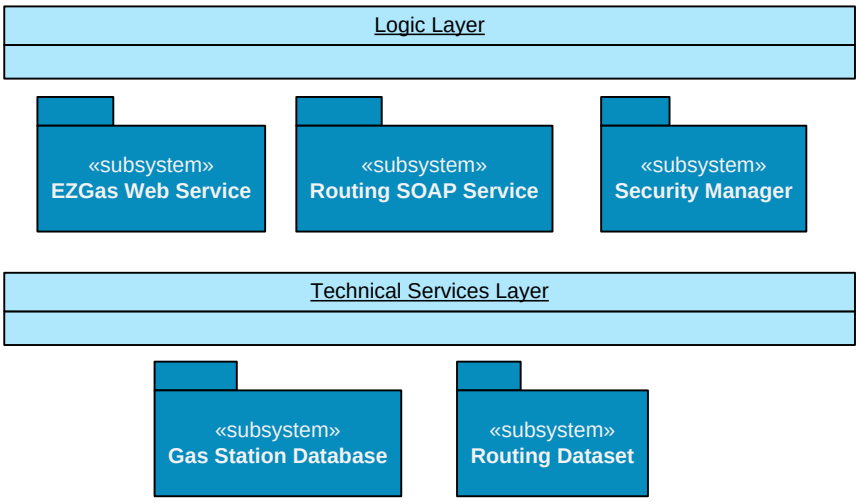


Figure 3.6: The server's two-tier logical architecture

participate in all cases which, lacking all originally, will go by the name *User*. In the diagram, use cases have been grouped by their functionality and shaded accordingly so, for example, green-shaded use cases are all related to vehicle profiles.

From this point onwards, the term "GS" means "gas station".

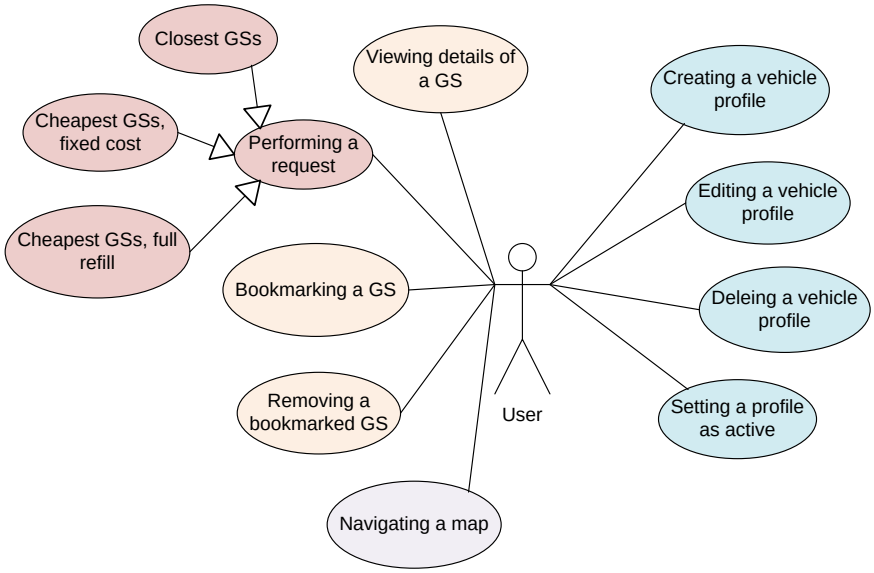


Figure 3.7: Identified use cases in EZGas

All previously defined use cases are now described in detail through the following use case tables. Each use case has been given a unique identifier made up of the string "UC" followed by a three-digit number.

3.3.1 Use cases related to vehicle profiles

These use cases are described in tables 3.2, 3.3, 3.4 and 3.5.

3.3.2 Use cases related to launching requests

These use cases are described in tables 3.6, 3.7, 3.8 and 3.9.

3.3.3 Use cases related to viewing gas station details

These use cases are described in tables 3.10, 3.11 and 3.12.

ID	**UC001**
Title	**Creating a vehicle profile**
Involved actors	*User*
Description	A vehicle profile is defined by the user in order to be able to make requests and, in general terms, use the application.
Preconditions	None.
Detailed steps	1. User enters profile data
Postconditions	A vehicle profile is created and available for its use.
Exceptions	• User enters invalid data.
Related use cases	UC002, UC003, UC004.

Table 3.2: UC001 - *Creating a vehicle profile*

ID	**UC002**
Title	**Editing a vehicle profile**
Involved actors	*User*
Description	The value of one or more fields of an stored profile are changed by the user.
Preconditions	The profile has been previously created.
Detailed steps	1. User selects the profile to edit 2. User modifies information at his/her discretion 3. User saves the profile
Postconditions	The profile is updated with the last changes.
Exceptions	• User enters invalid data.
Related use cases	UC001, UC003, UC004.

Table 3.3: UC002 - *Editing a vehicle profile*

ID	**UC003**
Title	**Deleting a vehicle profile**
Involved actors	*User*
Description	A vehicle profile is removed from the application.
Preconditions	The profile has been previously created.
Detailed steps	1. User selects the profile to delete
Postconditions	The profile is deleted.
Exceptions	• The profile to delete does not exist.
Related use cases	UC001, UC002, UC004.

Table 3.4: UC003 - *Deleting a vehicle profile*

ID	**UC004**
Title	**Setting a profile as active**
Involved actors	*User*
Description	A vehicle profile is set as active so that the application considers its parameters when serving user's requests.
Preconditions	The profile has been previously created.
Detailed steps	1. User sets the profile as active
Postconditions	The profile becomes the active one. The previously active profile is no longer so.
Exceptions	• The profile is already the active one.
Related use cases	UC001, UC002, UC003.

Table 3.5: UC004 - *Setting a vehicle profile as active*

ID	**UC005**
Title	**Performing a request**
Involved actors	*User*
Description	The user initiates a request to discover the best GSs around him, given specific calculation criteria.
Preconditions	A vehicle profile is set as active.
Detailed steps	1. User selects which type of request is to be performed 2. Client collects active profile's data and builds request 3. Server receives and parses request 4. Server calculates best matches according to request details 5. Server delivers results 6. Client processes results
Postconditions	The set of results is shown to the user.
Exceptions	• Server is unavailable. • User's location is unavailable. • Connection fails. • Request/response are malformed.
Related use cases	UC006, UC007, UC008.

Table 3.6: UC005 - *Performing a request*

ID	**UC006**
Title	**Requesting the closest gas stations around the user**
Involved actors	*User*
Description	The user initiates a request to discover the closest GSs around him.
Preconditions	A vehicle profile is set as active.
Detailed steps	1. User initiates request 2. Client collects active profile's data and builds request 3. Server receives and parses request 4. Server calculates best matches according to request details 5. Server delivers results 6. Client processes results
Postconditions	The set of results is shown to the user.
Exceptions	• Server is unavailable. • User's location is unavailable. • Connection fails. • Request/response are malformed.
Related use cases	UC005, UC007, UC008.

Table 3.7: UC006 - *Requesting the closest gas stations around the user*

ID	**UC007**
Title	**Requesting the best gas stations around the user given a fixed cost**
Involved actors	*User*
Description	The user initiates a request to discover which GSs around him deliver the highest volume of fuel according to a fixed purchase cost.
Preconditions	A vehicle profile is set as active.
Detailed steps	1. User selects the fixed cost to consider 2. Client collects active profile's data and builds request 3. Server receives and parses request 4. Server calculates best matches according to request details 5. Server delivers results 6. Client processes results
Postconditions	The set of results is shown to the user.
Exceptions	• Server is unavailable. • User's location is unavailable. • Connection fails. • Request/response are malformed. • User provides an invalid fixed cost.
Related use cases	UC005, UC006, UC008.

Table 3.8: UC007 - *Requesting the best gas stations around the user given a fixed cost*

ID	**UC008**
Title	**Requesting the cheapest gas stations around the user for a full refill**
Involved actors	*User*
Description	The user initiates a request to discover which GSs around him are the most economical for filling up his/her vehicle's tank.
Preconditions	A vehicle profile is set as active and a GS has been bookmarked.
Detailed steps	1. User selects the remaining amount of fuel in the tank 2. Client collects active profile's data and builds request 3. Server receives and parses request 4. Server calculates best matches according to request details 5. Server delivers results 6. Client processes results
Postconditions	The set of results is shown to the user.
Exceptions	• Server is unavailable. • User's location is unavailable. • Connection fails. • Request/response are malformed.
Related use cases	UC005, UC006, UC007.

Table 3.9: UC008 - *Requesting the cheapest gas stations around the user for a full refill*

ID	**UC009**
Title	**Viewing details of a gas station**
Involved actors	*User*
Description	Detailed information about a given GS is shown to the user.
Preconditions	A vehicle profile is set as active. A request has been carried out or at least one GS has been bookmarked by the user.
Detailed steps	1. User selects the GS whose details he/she desires to know.
Postconditions	Detailed information of the selected GS is shown to the user.
Exceptions	None.
Related use cases	UC010, UC011.

Table 3.10: UC009 - *Viewing details of a gas station*

ID	**UC010**
Title	**Bookmarking a gas station**
Involved actors	*User*
Description	User marks a given GS for later reference.
Preconditions	A vehicle profile is set as active.
Detailed steps	1. User bookmarks the desired GS.
Postconditions	The given GS is now bookmarked.
Exceptions	• The selected GS has been already bookmarked.
Related use cases	UC009, UC011.

Table 3.11: UC010 - *Bookmarking a gas station*

ID	**UC011**
Title	**Removing a bookmarked a gas station**
Involved actors	*User*
Description	User removes a given GS from the bookmarks list.
Preconditions	A vehicle profile is set as active.
Detailed steps	1. User selects which GS is to be removed from the bookmarks list.
Postconditions	The given GS is no longer bookmarked.
Exceptions	• The selected GS was not bookmarked before.
Related use cases	UC009, UC010.

Table 3.12: UC011 - *Removing a bookmarked a gas station*

3.3.4 Uncategorized use cases

These use cases are described in table 3.13.

3.4 Software requirements specification

This section presents all the identified software requirements, divided into categories according to their nature. Requirements are also divided according to whether they belong to the server or the client application.

Every requirement described in this section follows a specific naming convention, much like use cases in section 3.3. Such a convention is composed of a three-letter prefix followed by a three-digit numeral. The prefix in turn consists of a component identifier (1 letter) and a type identifier (2 letters). The component identifier can be either S or C, indicating that the requirement applies to the server or the client respectively; the type identifier can take any of the following values:

- FN: functional requirement
- PF: performance requirement
- SC: security requirement

3.4.1 Functional requirements

The following requirements are mainly related to the server application.

ID	**UC012**
Title	**Navigating a map**
Involved actors	*User*
Description	User navigates a map to discover nearby GSs.
Preconditions	A vehicle profile is set as active.
Detailed steps	1. User moves around the map. 2. Server sends a list of GSs which are within the map area the user is viewing.
Postconditions	New GSs are shown on the map.
Exceptions	None.
Related use cases	None.

Table 3.13: UC012 - *Navigating a map*

SFN001 The server shall expose a unique endpoint where all requests will be addressed.

SFN002 The exposed endpoint shall be a web service.

SFN003 The web service shall be able to receive multiple parameters through the web service URL query string.

SFN004 The server shall implement the following modes of operation to calculate the best set of gas stations for any given request:

- mode A: the server calculates the closest gas stations to the user
- mode B: the server calculates the set of gas stations where the highest volume of fuel can be purchased with a fixed cost
- mode C: the server calculates the most economical gas stations for filling up the user vehicle's tank size
- mode D: the server returns all gas stations around a center point and within a specific radius[2].
- mode E: the server delivers information of a specific gas station

SFN005 The server shall expect, as one of the parameters provided to the web service, which operation mode must be used.

[2]As similar as this mode sounds to mode A, there are slight differences which are treated in chapter 4

SFN006 The server shall also expect, as parameters provided to the web service, the following attributes:

- the size (in liters) of the vehicle's fuel tank
- the coordinates, as a latitude-longitude pair, of the user's position
- the type of fuel to consider during the calculation process
- the fuel consumption rate (in liters per 100 kilometers) of the vehicle
- the radius (in kilometers) of the area where seraches are to be carried out
- the intended cost (in €) of the refuelling operation
- the remaining amount of fuel (in liters) within the vehicle's tank
- the coordinates, as a latitude-longitude pair, of the gas station of which details are desired

SFN007 If, for a given request, any of the required parameters for the associated operation mode is missing, an error shall be returned.

SFN008 The server shall be able process requests for, at least, the following fuel types:

- unleaded 95-octane gasoline
- unleaded 98-octane gasoline
- standard diesel
- premium diesel
- biodiesel

SFN009 The server shall automatically update the full price list for all available gas types and gas stations.

SFN010 When operating with distances between two geographic points during its calculations, the server shall use the driving distance instead of the linear distance.

SFN011 If the driving distance is not available, the server shall fall back to the linear distance.

SFN012 The web service shall reply to requests in the JSON format.

SFN013 The server shall include within its replies the following information for each gas station:

- the gas station name
- its latitude and longitude
- the price per liter of each available fuel type
- its opening schedule
- the distance to the user
- the volume of fuel (where applicable) that can be refuelled
- the cost (where applicable) of the operation

The following requirements primarily affect the client application.

CFN001 The client application shall be able to build and send requests for any of the available operation modes exposed by the server.

CFN002 The client application shall be able to parse server replies and display results to the user appropriately.

CFN003 If the client application attempts to parse a malformed server reply the reply shall be discarded and an error message shall be traced.

CFN004 The client application shall automatically obtain the user's location from different location providers which are, in order of preference, the following ones:

- the device's GPS antenna
- a WiFi access point
- the mobile network cell

CFN005 If a given provider is not available, the client application shall attempt to obtain the location from the next provider in the list.

CFN006 If no provider is available, the application shall abort the operation and display an error message.

CFN007 The client application shall allow the user to tag any gas station as favorite. A reference to favorited gas stations shall be kept by the client application to allow their later review by the user.

CFN008 The client application shall allow the user to create, modify and delete vehicle profiles.

CFN009 A vehicle profile shall be composed of a name, a fuel type, the vehicle's fuel consumption rate and its fuel tank size.

CFN010 If the user attempts to create a vehicle profile with invalid data, the client application shall abort the operation and appropriately inform the user.

CFN011 The client application shall allow the user to set one of the existing profiles as the active one. No more than one profile shall be active at the same time.

CFN012 If no active profile is set, the client application shall prevent the user from launching requests to the server.

CFN013 If the user tags a profile as active, the previously active profile, if any, shall be untagged.

3.4.2 Non-functional requirements

Performance requirements

SPF001 The web service shall take no longer than 15 seconds to reply to any request which uses the maximum possible search range.

SPF002 The web service shall be able to support a considerably large number of concurrent requests without disrupting its performance.

CPF001 The client application shall be able to function on any device running version 2.1 or greater of the Android operating system.

Security requirements

SSC001 The web service shall implement a security mechanism for denying requests which are not authenticated and/or have expired.

SSC002 The security mechanism shall consist of a pair of authentication parameters which are provided to the web service alongisde the rest of parameters specific to the request.

SSC003 One of the authentication parameters shall be a UNIX timestamp which specifies the time at which the request was issued.

SSC004 The other authentication parameter shall be a hash string, which is built by applying a series of hash functions to a string composed of the timestamp and a secret key.

SSC005 A request shall be considered expired if the difference of the server's time at the moment of receiving the request and the issuing time is greater than a given threshold.

SSC006 A request shall be considered invalid if, using the provided timestamp, the server is unable to produce the same hash string as the provided one.

SSC007 For a request to be accepted, it must have not expired and must be valid.

SSC008 If a request fails to pass the security mechanism, the server shall reply with an error.

Chapter 4

Implementation

This chapter will describe the process of taking the software requirements, which were previously defined, to a completely implemented, fully working software solution. The following sections will provide some detailed insight into the architectural decisions taken throughout the project lifetime as well as information regarding the development tools used. A series of diagrams will illustrate how all components are internally structured and how they interact with each other.

Please note that, for the sake of simplicity, class diagrams will not include getter and setter methods even though most of them, if not all, are actually implemented.

4.1 Programming languages

As discussed in section 2.2, there is a vast array of programming languages from which to choose when developing software. Every language presents its own benefits and drawbacks, and there is no absolute perfect solution which fits every project. Thus some research is required in order to identify the best candidates for a given scenario considering its circumstances as well as implementation and economical aspects, among others.

This section will present the choices taken by the developer as well as some reasoning about why they were thought to be the best ones. Since EZGas is client-server application, a distinction will be made regarding this architecture.

4.1.1 Server side languages

EZGas exposes its calculation facilities by means of a web service which must carry out some logic and at the same time check and store all the available fuel prices. Regarding the server-side scripting there are a handful of options, among which PHP, Java, ASP.NET or Python stand.

From all the above, ASP.NET was discarded because, being a .NET language, is built on Microsoft's Common Language Runtime and therefore needs the .NET Framework[29]. This means that code written in ASP.NET can only run on a Windows box and, being a commercial product, requires the purchase of software licenses which are likely to increase the project's costs.

Python is a general-purpose language which allows several paradigmns, like object-oriented programming. It is a free language with a wide range of applications, one of them being used at server-side scripting, and is used by over a million users[31]. As powerful and valid as Python is for developing an application such as EZGas, the main reason it was not considered was the author's lack of familiarity with it. Python is considered an easy-to-learn language, but having to learn it from scratch is a significant risk to the project considering its short deadlines.

This leaves Java and PHP alone for the final decision. Both languages were already described in section 2.2. However, for the purposes of this discussion it must be stated that both were considered good fits as the two of them are free, popular languages with huge community support[16][13] and the author is comfortable working with any of them.

In the end, PHP was the chosen programming language for the server side as it allows for faster prototyping, does not need to recompile code after every change (as it is an interpreted language) and web hosting providers offer cheaper prices for PHP-based services[24] than Java-based ones[23]. It must be noted, however, that Java might have been considered instead if EZGas was a larger, more complex project.

For what concerns data storage three candidate RDBMS were considered, them being Oracle, MySQL and PostgreSQL. Just as with ASP.NET, Oracle was ruled out due to its commercial approach[1].

Regarding MySQL and PostgreSQL it is worth noting that, even though some years ago both systems presented clearly-defined advantages and drawbacks, nowadays they are very similar both feature- and performance-wise. At first PostgreSQL was the favored selection due to its spatial database extension, PostGIS, which was meant to provide the driving distance calculations required by EZGas for improved accuracy. However, once an easier solution was found (see osm2po in section 4.2), MySQL was the adopted system mainly because it is the usual RDBMS featured alongside PHP and Apache by most web hosting providers.

4.1.2 Client side languages

The client application is targeted at Android devices. This fact automatically limits the available languages, as Android applications are to be written

[1] Funnily enough, MySQL also belongs to Oracle Corporation as the company (actually, Sun Microsystems) purchased MySQL AB back in 2008[10].

in Java[19][5]. The Android platform also provides a way of running C and C++ code through the use of its Native Development Kit (NDK); howver, this only benefits certain types of applications (specifically those which are CPU-intensive or when a large legacy codebase already exists) and it always comes at the expense of increasing the application's complexity[27].

EZGas does not match this profile as all heavy calculations are performed on the server side, basically leaving the client with presentation logic only; this is why just Java is deemed appropriate. Furthermore, Java is not too complex and the author is already familiar with it, therefore the learning process is expected to be quite easy and take a small amount of time. This is likely to have a positive impact on the project's costs as well.

Even though EZGas features very modest local storage requirements, the application's shared preferences are not enough for that purpose; this is why some other type of local persistence is required. SQLite is a lightweight, transactional database engine[37] natively supported by Android and, as such, is the best solution for such an scenario.

4.2 Tools

There are a number of tools which have been used to aid the developer during the implementation of EZGas. This section presents and gives a small insight into each one of them.

4.2.1 Apache Web Server

Apache has been the most popular HTTP server in the world since 1996 and, as such, is part of the most common offers by web hosting providers alongisde PHP and MySQL in what is known as the *LAMP stack*. It can run on a large number of systems among which Windows and Linux can be found, and there are lots of compiled modules which add extra functionality such as URL rewriting, SSL support, authentication, compression or virtual hosting.

It has been chosen in favor of other alternatives such as Microsoft's Internet Information Services (IIS) due to the former being free, open source and proven to be reliable and secure.

4.2.2 Subversion

Subversion, also known as SVN, is a tool which belongs into the category of the so-called *version control systems* (VCSs). A VCS is a tool which manages files and directories and tracks all changes made to them throughout time[39], usually in a distributed, across-the-network manner.

SVN was created by CollabNet in 2001 as a replacement for Concurrent Versions System (CVS), an early VCS[39] which was starting to be consid-

ered an obsolete system. SVN It is a free competitor of other commercial VCSs such as Perforce or ClearCase.

The way SVN works is by managing a centralized repository to which users connect and read files. By *reading* files, users actually download a copy of the latest version into a local directory called *working copy*, that is, they *check out* those files. Users make changes to their local working copies and, once they are finished, they publish their modifications back to the central repository by *committing* them[39]. Every time a commit is carried out, the new state of the repository is considered a new *revision*.

Up to this point SVN looks like a regular file system. However, being a VCS, there are a couple of feature which turn SVN into an awesome collaborative-editing tool.

First, SVN keeps track of every committed revision. By doing so, users can go back, or *revert*, to a previous revision if considered appropriate. Second, if two users modify the same file at the same time and attempt to commit later on, SVN will compare both commits and attempt to *merge* their contents and keep the file up to date.

SVN was used in EZGas to maintain its code under control and be able to:

- develop the application in different computers, as long as the repository is reachable,
- roll back to a previous version in case of a considerably large mistake, and
- keep track of when a bug first appeared or when it was corrected

4.2.3 Eclipse

Eclipse is an Integrated Development Environment (IDE) created by IBM in 2001 and, originally, aimed to code in Java. However, since it features a powerful plug-in system, third-party extensions allow Eclipse to be turned into an IDE for a number of other popular languages such as C/C++, Ruby, PHP or Python. In fact, the Java IDE is itself a plug-in (JDT, Java Development Tools). Furthermore, there is a huge set of extensions which are not related to any specific programming language but to other tools and utilities, such as the Subclipse plug-in (which incorporates a direct interface to SVN functionality right into Eclipse).

Eclipse also provides advanced functionality such as interactive and automated refactoring which greatly helps developers to maintain modest-and-up codebases.

Eclipse is currently managed by the Eclipse Foundation and its current latest stable release is Eclipse 3.7 *Indigo* while 3.8 *Juno* is expected to be released any time soon[15].

4.2.4 Android SDK

The Android Software Development Kit (SDK) is a large set of libraries and other resources and tools which stand on top of the Android platform[19].

On one hand, the libraries provided by the SDK comprise a series of Java classes which are to be used by developers when developing Android applications. Some of them are strictly fundamental, such as the `Activity` class when developing a UI-based application, whereas some others provide functionality which may or might not be used by the programmer, such as the `LocationManager` class, which is used to obtain the user's location from one of the device sensors.

Apart from the libraries, the SDK also provides a set of tools which are meant to help the programmer in development tasks. Such a tool is the Android emulator, which is a sort of virtual machine (based on the QEMU technology[19]) which runs the very same Android operating system and mimics the behavior of a real device but runs on regular computers, allowing most users to fully develop and test their applications without even owning an actual Android phone. These great features, however, come at the cost of speed. This is why developers usually prefer to test their programs on a real device if possible.

Android Virtual Devices are the virtual machines which have just been described and which run on the Android emulator. AVDs can be configured using the AVD Manager, also bundled with the SDK, and users can create several AVDs each targeting a different platform version. This way they can test how their applications behave in new and old devices at no additional cost.

There is a debug tool named Android Debug Bridge and shortened *adb*. Adb is used to provide connectivity between a device and the development environment and allows the user to install and launch applications, explore the device file system and even run a subset of Linux commands.

The Dalvik Debug Monitor Server is a debugging tool which allows developers to monitor all applications running on a device[19]. Through the use of DDMS one can get screenshots from the device, monitor application threads, access real-time logging, analyze memory leaks and even cause the Garbage Collector to act on a given process and collect memory usage statistics.

Finally, all the previous tools can be easily integrated into Eclipse by using the Android Development Tools plug-in (ADT). The ADT allows users to access the DDMS, the Logcat or the AVD Manager directly from within Eclipse and usually as individual perspectives[19].

4.2.5 Notepad++

Notepad++ is a free and open source Windows-based text editor which intends to be a replacement for Windows' native Notepad. It features a vast number of improvements such as syntax highlighting, regular-expression search and replace, multi-view and autocompletion among others[22], and these features can be extended by the use of third-party plug-ins.

Notepad++ can be seen running in figure 4.1.

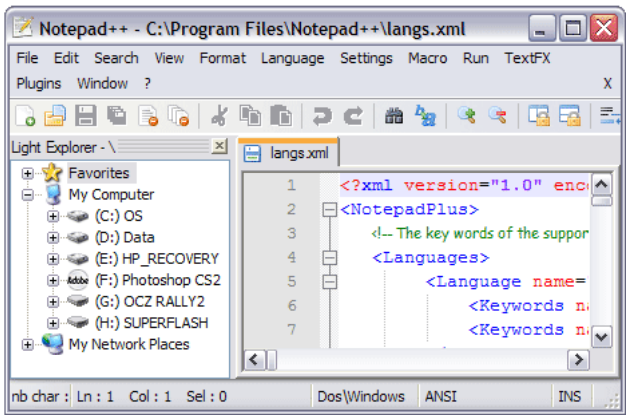


Figure 4.1: Notepad++

4.2.6 phpMyAdmin

phpMyAdmin is a web-based administration interface for managing MySQL databases which was first released in 1998. It is currently a free and open source project hosted at SourceForge.net and provides very similar capabilities than other administration software such as the official, desktop application MySQL Workbench.

During the development of EZGas, phpMyAdmin was used to create, edit, export and test the database used by the server application in a much quicker way than if the original command line tool was used.

4.2.7 osm2po

Osm2po is a tool which works with an OpenStreetMap data file (discussed previously in section 2.3) and serves a twofold purpose:

- converts OpenStreetMap data into a custom binary format file, and
- works as a routing engine using said binary file

Once the conversion is done, osm2po loads the custom binary file into memory and starts a SOAP-based web service on the background which is able to calculate best routes between to coordinates in space. This web service is key to EZGas as it is used to quickly obtain the real driving distance between two points instead of the straight distance, meaning more accurate results. Osm2po also features a fully-working web server with a graphical interface for calculating such routes, but it is not used in EZGas.

4.3 Server side implementation

4.3.1 Database

EZGas needs a database to store the latest available information regarding all gas stations in Spain. The database itself is simple and presents the schema shown in figure 4.2.

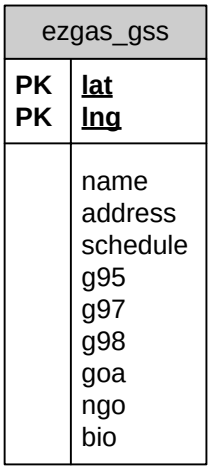


Figure 4.2: EZGas' server database schema with its only table

A PHP class, `DbGateway`, was written to provide an abstraction layer over the database. Its definition is shown in figure 4.3.

4.3.2 Updater module

The price data source updates its price information during the night, shortly after midnight. A specific component of EZGas, the updater module, is in charge of fetching all price files from the source and reading them to update its own database.

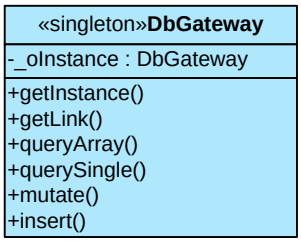


Figure 4.3: The DbGateway class

Data comes in a .zip file whose contents are decompressed and extracted into a temporary location. The extracted file is a custom-formatted CSV file which contains entries looking like the follwoing samples:

```
2.10873, 41.35464, "MEROIL L-D: 24H 1,360 e"
2.67292, 41.72567, "PETROCAT L-D: 06:00-22:00 1,369 e"
```

After inspecting some more entries, a pattern is extracted as if it were a grammar. It is shown in figure 4.4.

The updater module, by means of class `FuelPriceImporter`, updates prices of a single fuel type each time, repeating the same steps in every iteration.

First, a backup of the current data is made to assure its recovery should anything go incredibly wrong during the process. Then a new temporary table is created which will receive the new prices and will be swapped with the old one once the process finishes. Second, the price file for the appropriate fuel type is downloaded by class `FuelPriceDownloader`. Then the CSV file is read by class `FuelPriceCsvParser` and a temporary structure is created on memory which facilitates data retrieval. Lastly, prices are inserted into the temporary table and later swapped with the old table. This guarantees that old prices are still available to users even while the system is updating, even though this process only takes 5 to 10 seconds.

Classes belonging to the updater module are described in figure 4.5.

In order to repeatedly run this task at regular intervals, as specified in subsection 3.4.1 a periodic task needs to be configured on the server's oper-

```
ENTRY    ::= LAT,LNG,DETAILS
LAT      ::= <decimal number>
LNG      ::= <decimal number>
DETAILS  ::= "NAME SCHEDULE PRICE"
NAME     ::= NAME{ NAME}*
NAME     ::= <uppercase string>
SCHEDULE ::= WEEKDAYS: TIMES
PRICE    ::= <decimal number> e
WEEKDAYS ::= DAY-DAY
DAY      ::= L|M|X|J|V|S|D
TIMES    ::= 24H|TIME-TIME
TIME     ::= HOUR:MINUTE
HOUR     ::= <from 00 to 23>
MINUTE   ::= <from 00 to 59>
```

Figure 4.4: The grammar describing gas station entries within the CSV file

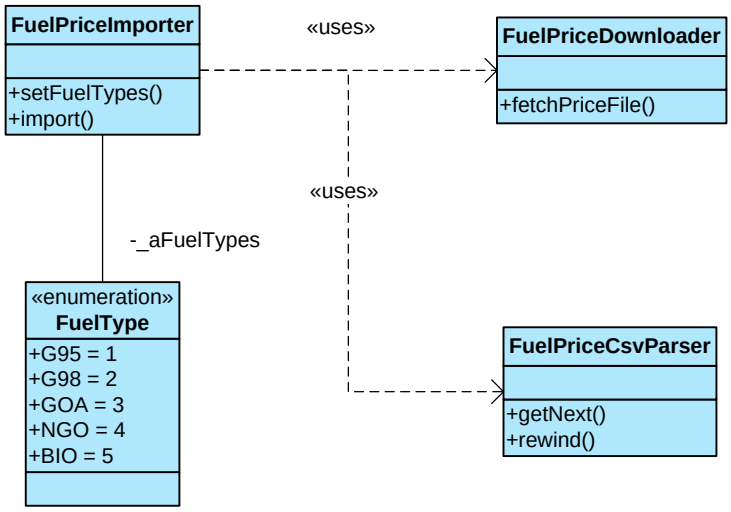


Figure 4.5: Class diagram of the updater module

ating system. The author's implementation runs on a Linux box, therefore this is achieved by means of a *cronjob*. Such a job is described in crontab as follows[45], where *update.php* is the PHP script which kickstarts the process:

```
0 1 * * 1-5 /var/www/update.php
```

4.3.3 Web service

To properly detail how the web service is built, let's analyze the path followed by an arbitrary request from the moment it reaches the server.

The endpoint script makes use of the `ServiceRequestFactory` class by passing it all the parameters provided in the request's URL. The factory class inspects each parameter, determines which ones are required for the given request type and validates all of them. If parameters are correct then the `ServiceRequestAuthenticationManager` class will be provided with the authentication parameters and will determine whether or not the request is valid and thus is allowed to go on.

If the process is successful the factory will return an instance of the `ServiceRequest` class and the endpoint will use it as a convenient way of accessing all parameters it considers appropriate (rather than directly accessing the URL query string). If any error arises during the parsing process a subtype of `ServiceRequestException` will be thrown (for example, if the operation mode parameter is missing the subclass will be `InvalidRequestCodeException`, whereas if the request fails to be authenticated the exception will be of type `UnauthorizedRequestException`).

Figure 4.6 illustrates the aforementioned elements.

Moving on within the request processing, the next step is to initialize the routing engine which will be used to calculate driving distances between any two geographical locations. As exposed in section 4.2, EZGas will rest on the *osm2po* tool which exposes its functionality by means of a SOAP service, this is why a series of classes were designed to connect with it: `Router` and `RoutingEngineSoapGateway`. The latter is the class which actually interacts with the service, whereas the former was created to provide some abstraction from the routing service itself (which might be helpful if the routing engine changes in the future). Both are described in figure 4.7.

The web service will first create an instance of the gateway and then will provide this instance to the Router constructor. From this moment onwards the service just needs to call the `getShortestPathDistance` function to retrieve the shortest driving distance between to points (the other function, `getClosestVertexToCoordinates`, is used to determine the elements in the routing graph which are closest to both the origin and destination).

The next step is to start the actual calculations, but this depends upon what type of request is being treated (refer to the *operation modes* described in subsection 3.4.1). However, all of them will at some point need to access the database and fetch one or more entries. To avoid a direct interaction with

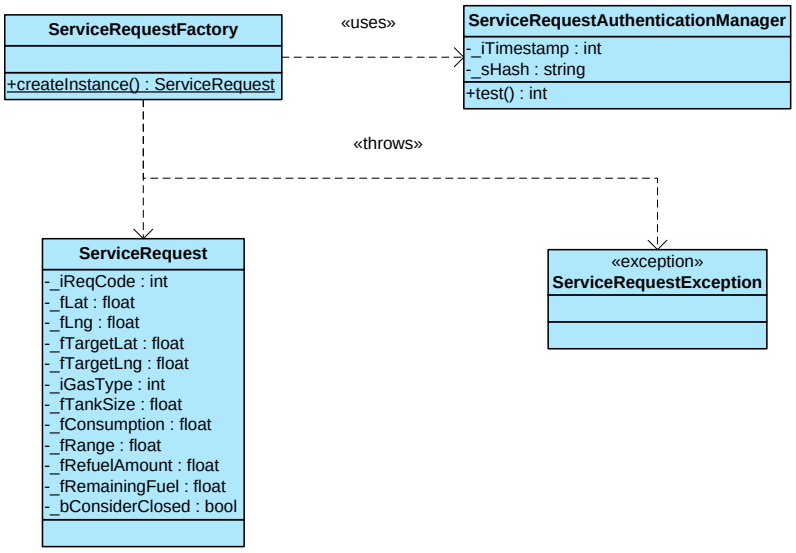


Figure 4.6: Classes related to a web service request

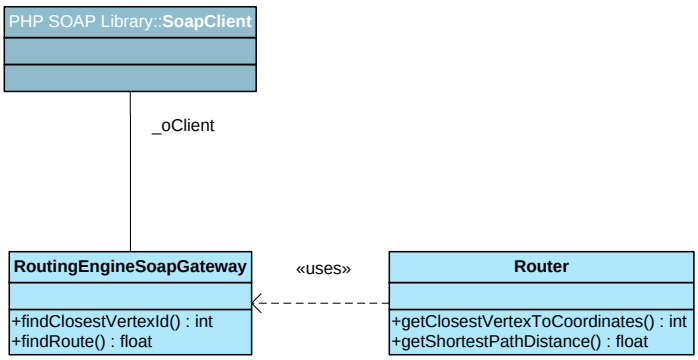


Figure 4.7: The Router and RoutingEngineSoapGateway classes

the MySQL driver, and to reduce lines of code, the application domain was modeled using two classes: `GasStation` and `GasStationSchedule`, shown in figure 4.8.

The `GasStation` class is the object-oriented representation of a gas station entity stored within the database but also features static functions which allow the retrieval of those same entities by following the Active Record pattern.

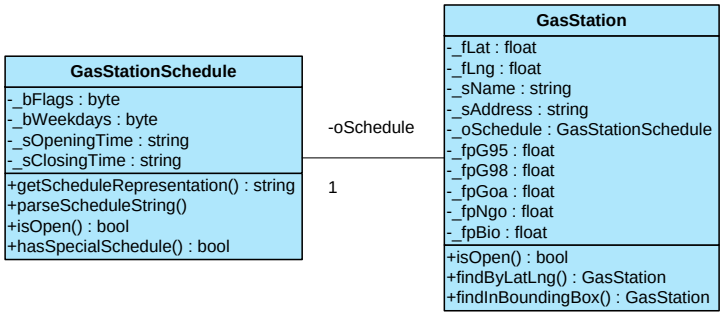


Figure 4.8: The GasStation and GasStationSchedule classes

The `findByBoundingBox` function is used by the service when conducting a search in some geographical area: since there is a limit on how far away from the user the server should look for gas stations, this function helps by performing a first filter on the set of result candidates and helps speeding up the processing. This way, the server will only have to examine tenths of gas stations instead of the full set of about 8,000 elements.

This is graphically described in figure 4.9, where the bounding box is the green diamond shape. All gas stations within the bounding box will be returned by `findByBoundingBox` (green circles), whereas any other gas station will be discarded (blue circles). Note how further filtering is required since some results are within the bounding box but out of the search range.

If the service wishes to retrieve a specific gas station then the `findByLatLng` function will be used.

The server, having a set to work on, will iterate through each gas station and will calculate a *key value* depending upon the request type (of course, if the request asks for a specific gas station there is nothing to iterate through). Key values will be used to sort the result set later on.

- if the request is asking for the closest gas stations, the key value is the distance between the gas station and the user

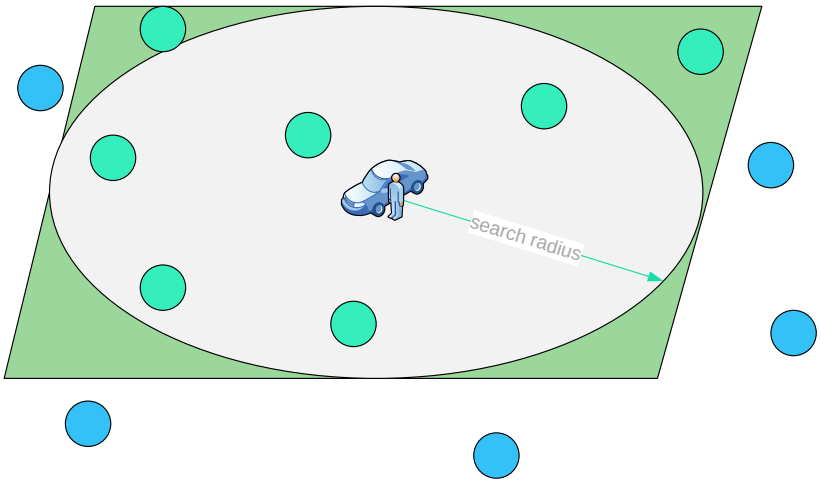


Figure 4.9: An example illustrating the usage of a bounding box to filter results from the database

- if the request is asking for the cheapest gas stations given a fixed cost, the key value is the volume of fuel that can be purchased at that cost minus the volume of fuel that will be consumed on the trip to the gas station
- if the request is asking for the cheapest gas stations to top up the tank, they key value is the price of filling the tank up to its full capacity plus the price of the trip to the gas station

The previous descriptions are represented in equations (4.1), (4.2) and (4.3) respectively. In those formulas, d_i is the shortest driving distance between the gas station and the user, c is the fixed cost, p is the fuel price per liter, s is the fuel tank size, r is the consumption rate of the vehicle and q is the remaining amount of fuel inside the tank (between 0 and 1, 0 meaning empty and 1 meaning full).

$$k_i = \text{distance} = d_i \tag{4.1}$$

$$k_i = \text{refuelled volume} - \text{consumed volume} = \frac{c}{p} - \frac{d_i r}{100} \tag{4.2}$$

$$k_i = \text{refuelling cost} + \text{travelling cost} = sp(1-q) + \frac{d_i c}{100p} \tag{4.3}$$

Up to this point, the only thing left to do is sorting the results by the appropriate key value and output the response. The `ServiceResponse` class is a convenient way of adding results to a result set. It will also automatically format the set as a JSON string by means of the `toJson` function, which belongs to the `IJsonable` interface. This is illustrated in figure 4.10.

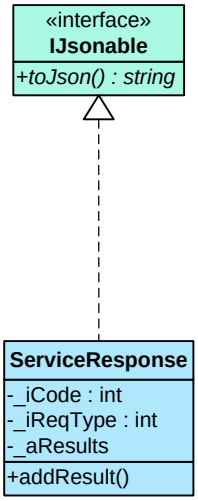


Figure 4.10: The ServiceResponse class and the IJsonable interface it implements

Sample response

Responses from the web service will be formatted as a JSON string. A sample output is presented in figure 4.11.

Web service reference

The web service is accessed by means of a HTTP GET request addressed at the endpoint URL. The parameters are provided inside the query string, so it would look like the following:

http://myserver/endpoint.php?p1=a&p2=b&p3=c

Table 4.1 describes all the possible parameters and whether or not they are required depending upon the type of request. *R* indicates that the parameter is required in the given operation mode, *O* means that it is optional and a blank space means that the parameter is not used in such mode. See subsection 3.4.1 for more info about the aforementioned modes.

```
{
  "_iCode": 0,
  "_iReqType": 1,
  "_aResults": [
    {
      "_oGs": {
        "_fLat": "40.54604",
        "_fLng": "-3.41067",
        "_sName": "REPSOL",
        "_sAddress": "Calle Llesquche 24, Pinto, Espa\u00f1a",
        "_fG95": "1.386",
        "_fG98": "1.505",
        "_fGoa": "1.302",
        "_fNgo": "1.363",
        "_fBio": null,
        "_oSchedule": {
          "_btFlags": 0,
          "_btWeekdays": 64,
          "_aOpeningTime": [
            5,
            0
          ],
          "_aClosingTime": [
            22,
            0
          ]
        }
      },
      "_fCost": null,
      "_fVol": null,
      "_fDist": 1.7601212148
    }
]
}
```


Figure 4.11: An example response from the server

Name	Description	Mode A	Mode B	Mode C	Mode D	Mode E
t	Timestamp	R	R	R	R	R
h	Hash	R	R	R	R	R
c	Request code	R	R	R	R	R
lt	User latitude	R	R	R	R	R
ln	User longitude	R	R	R	R	R
tt	Target latitude					R
tn	Target longitude					R
gt	Gas type	R	R	R	R	
ts	Tank size		R	R		
cn	Consumption rate		X	X		
rn	Search range	O	O	O	O	
pr	Refuelling cost		X			
rc	Remaining fuel			X		
cc	Consider closed GSs	O	O	O	O	O

Table 4.1: Expected parameters by the web service

4.4 Client side implementation

The Android application follows a three-tier design which includes a presentation layer (the *Activities*), a logic layer and a persistence (or technical services) layer. This section will review each tier from the lowest- to the highest-level one.

4.4.1 Technical services tier

In order to satisfy its storage requirements, the client application makes use of two components for keeping data locally: shared preferences and an SQLite database. The former store key-value pairs and are used for saving users' settings and preferences, whereas the latter is a lightweight, self-contained database system which, as described in section 4.2, is natively supported by the Android platform with no external dependencies.

The database is used to store all vehicle profiles defined by users as well as their bookmarked gas stations, and features the schema presented in figure 4.12.

In order to access these tables the application requires some sort of abstraction layer. The client implements this layer in a different way than the server: instead of using the Active Record pattern, there is one additional class per each domain object: `VehicleProfileDAO` and `FavoriteGasStationDAO`, both residing inside the `es.uc3m.tfg.ezgas.persistence.dao` package.

The DAO classes relay on a third class, `EZGasDbHelper`, which extends Android's native `SQLiteOpenHelper` and which is in charge of creating, updating and opening the database as required. The `EZGasDbHelper` class is located inside the `es.uc3m.tfg.ezgas.persistence` package.

If the application requires content to be retrieved from or inserted into the database it will call the DAO classes, which provide static methods that are specific to the model they represent. For example, `FavoriteGasStationDAO`

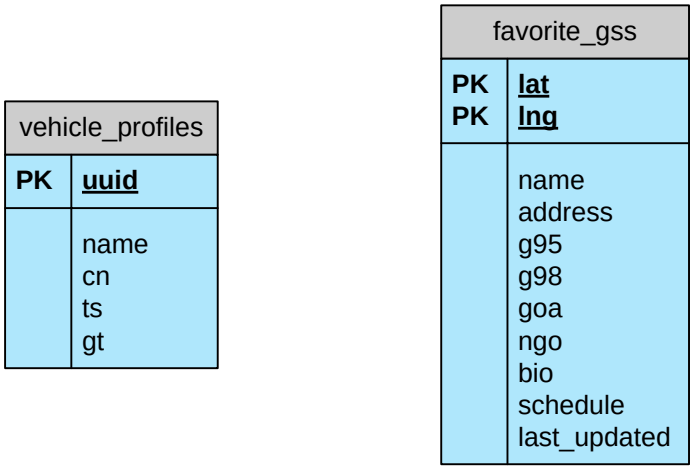


Figure 4.12: The client's database schema

has the `findByLatLng` method, which allows to retrieve a specific gas station form the database by its latitude and longitude.

Classes belonging to the persistence layer are represented in figure 4.13.

4.4.2 Logic tier

Domain classes

Classes which model the application domain are similar to those in the server, but with some additions. The package `es.uc3m.tfg.ezgas.model` features the Java implementation of the `GasStation` and `GasStationSchedule` classes, which are practically identical to those on the server side. However, two new classes exist: `VehicleProfile`, which represents user-created profiles, and `FavoriteGasStation`, which represents those gas stations which have been tagged as favorite by the user.

`VehicleProfile` is a pretty simple, independent and standalone class, whereas `FavoriteGasStation` extends the `GasStation` class and implements a new field representing the last time its information was updated. Since entities representing both classes are stored in the SQLite database, instances of these types can be created through the use of the DAO classes present in the technical services tier.

A graphical description of all classes belonging to this group is presented in figure 4.14.

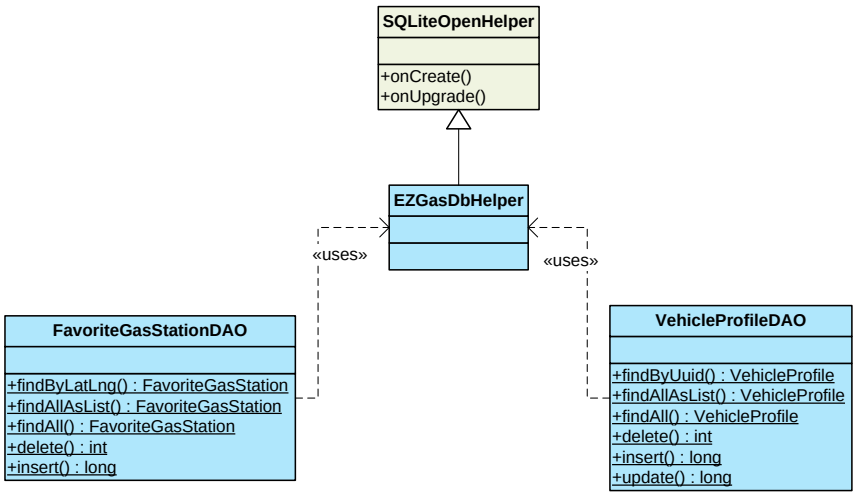


Figure 4.13: Persistence layer classes

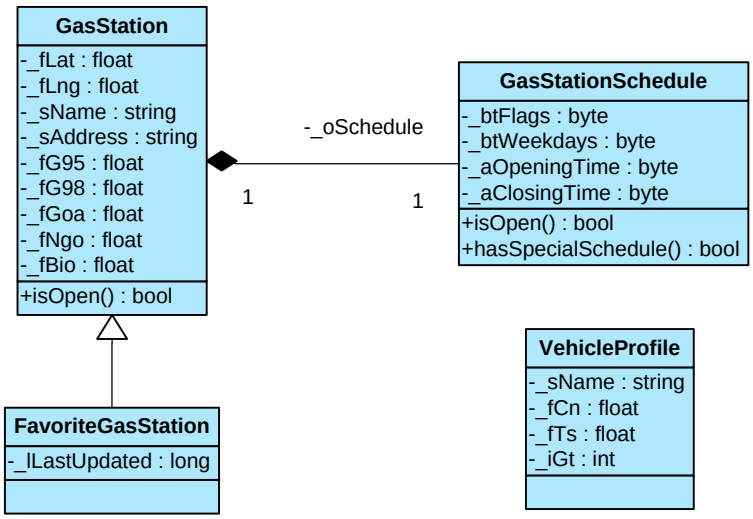


Figure 4.14: The client's domain classes

Service-related classes

In order for EZGas to be able to interact with the server a series of classes were implemented. Specifically, the package `es.uc3m.tfg.ezgas.service` contains, as illustrated in figure 4.15, all the classes which deal with creating and sending requests and processing responses.

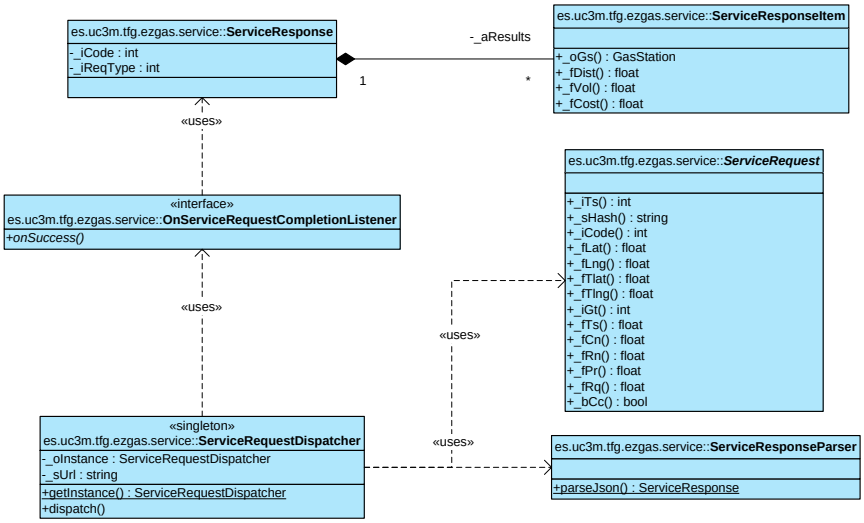


Figure 4.15: Web serice-related classes within the client application

Whenever the client needs to send a request to the server, it will start by creating a subclass of `ServiceRequest` (there is one per each type of request, such as `ClosestStationsRequest`) and populating its parameters. The next step is to obtain an instance of the `ServiceRequestDispatcher` class and providing the server URL to it. This class includes a method, `dispatch`, which will receive the previously created request object and also expects an implementation of the `OnServiceRequestCompletionListener` interface, which will be used as a callback once the request is replied by the server.

The `dispatch` method basically sends an HTTP GET request to the server, waits for the response and parses it using the `ServiceResponseParser` class, which in turn yields an instance of the `ServiceResponse` class. This instance will be provided to the callback's `onSuccess` method, which is expected to continue with the execution flow as it wishes (for example, presenting the results on a list).

The returned response contains a list of `SerivceResponseItem` objects, which themselves are composed of the key values calculated by the server (cost, volume and distance) and also of the *GasStation* instance they are

related to.

Utility classes

In addition to the classes described above, the logic tier also includes some other classes which were created for convenience purposes. For example, the `EZGasConstants` class is a method-less unit which is full of attributes declared as `static` and `final`. Those attributes contain information which is used throughout the client application and include, for example, default values for server requests and keys for accessing shared preferences.

4.4.3 Presentation tier

This layer is composed of all classes which are directly related to the graphical user interface. Within the Android platform such classes extend from the `Activity` class or one of its subclasses and, in the case of EZGas, are located within the `es.uc3m.tfg.ezgas.activities` package. Their content is similar in all of them as they mainly initialize the *widgets* (UI elements), process user input and start other activities as required.

EZGasMainActivity

This is the application entry point. The interface looks like a dashboard where the user can choose what to do. Tapping on any of the first three blue icons will send a request to the server and return the results in a list, whereas the fourth icon will start the map navigation activity. From this activity users can also see their bookmarked gas stations, edit their profiles and access the application settings.

This activity is shown in figure 4.16.

EZGasResultListActivity

Once the server replies to a request, the returned results (if any) will be displayed in this activity, as figure 4.17 shows. Each result includes the gas station name, its address and the calculated key value which depends upon the type of request.

Results are ordered from the best to the worst option and key values are shaded in five different colors, from light green to dark red, indicating how good a choice they are. If the user taps on any item, a new activity will be launched and details about the selected gas station will be displayed.

EZGasGsDetailsActivity

This activity presents detailed information about a gas station such as its name, address, opening schedule and the latest price data for all fuel types.

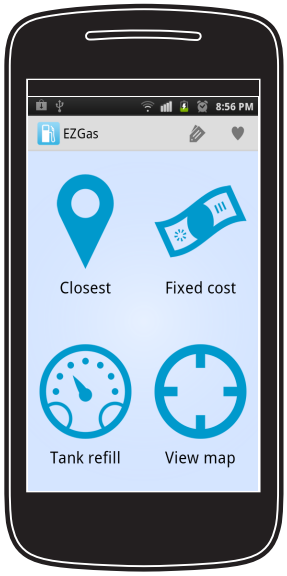


Figure 4.16: EZGasMainActivity

Figure 4.17: EZGasResultListActivity

It also allows the user to see the location of the given gas station overlaid on a map.

Devices running Android 3.0 and above allow the user to navigate between both views by using tabs and *Fragments*, whereas older versions make use of two different activities, `EZGasCompatibleGsDetailsActivity` and `EZGasGsOnMapActivity`.

If available, users can launch the Google Navigation app from within the factsheet view in order to get driving instructions to the gas station. Both sections can be seen in figure 4.18.

(a) The gas station factsheet interface

(b) The location of a gas station printed on a map

Figure 4.18: Detailed information about a specific gas station

EZGasMapActivity

The fourth big blue icon on the main dashboard will start this activity, which allows users to navigate through a map by swiping their finger. Every time the area of the map being displayed is changed, the server will reply with a set of gas stations being near the current center of the map and they will be displayed as icons. Tapping any icon will start the detailed information activity, as explained previously. Figure 4.19 shows the appearance of this activity.

The amount of gas stations being asked to the server is automatically adapted depending upon the current zoom level in order to save bandwidth, as there is no point in requesting all gas stations in a 10 km. radius if the map view only spans ten or twenty city blocks.

Figure 4.19: Gas stations being shown on a map

EZGasProfileListActivity

This activity lists all vehicle profiles defined by the user and allows to delete or update any of them and create new ones. If a new profile is created, the `EZGasNewProfileActivity` will be launched. Both activities are shown in figure 4.20.

EZGasFavoriteGasStationListActivity

This activity, displayed in figure 4.21 lists all gas stations which have been tagged as favorite by the user. It also allows to delete them if desired.

EZGasSettingsActivity

Users can edit some application preferences from this activity; namely, the server URL, the search range and whether or not to include closed gas stations in server responses. It also presents some basic information about the application such as the author and the version. Figure 4.22 shows a screenshot of this activity.

(a) EZGasProfileListActivity (b) EZGasNewProfileActivity

Figure 4.20: Profile management activities in EZGas

Figure 4.21: EZGasFavoriteGasStationListActivity

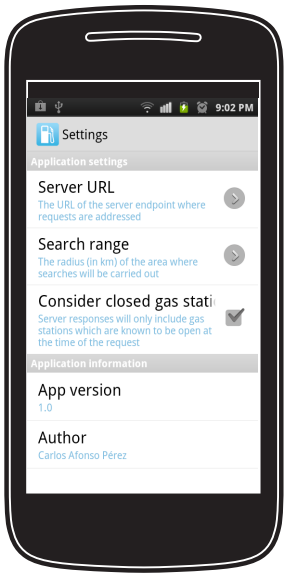


Figure 4.22: EZGasSettingsActivity

Chapter 5

System verification

Once the system is finally implemented it needs to be tested in order to verify that the specified requirements are met and that it works as expected. This chapter briefly details the procedures used to accomplish said tasks.

5.1 Test environment

The server side of the project was run on a computer powered by a 3.2 GHz Intel Pentium 4 processor with 1 GB of RAM and access to the Internet through an ADSL link. The computer ran Ubuntu 11.04 *Natty Narwhal* and the following software versions:

- PHP 5
- MySQL Server 5.5
- Apache Server 2.2
- osm2po 4.4.4.a

The client application was mainly tested on the following devices:

- a Samsung Galaxy S Plus running Android 2.3 (Gingerbread),
- an Android Virtual Device running Android 4.0 (Ice Cream Sandwich) and
- another Android Virtual Device running Android 2.1 (Éclair)

Additionally, some tests were carried out on a Samsung Galaxy Nexus also running Ice Cream Sandwich. All devices had a working Internet connection.

Most of the tests were run on the client application. However, some tests required to be executed directly on the server due to its nature (for example, passing invalid parameters to the server can't be done from the client as it automatically prevents this from happening).

5.2 Test results

First, the web service was tested in two ways: regular behaviour and exception handling.

The former was tested using the client application by sending several requests of varying types and with different parameters. This checked that the service responded as expected within a reasonable amount of time when requests were well formed and parameters were all within valid bounds.

These tests were used as well to check that the client application built requests and processed responses as specified. The overall involved requirements are: SFN001, SFN002, SFN003, SFN004, SFN005, SFN006, SFN008, SFN010, SFN012, SFN013, SPF001 and SSC007 for the server side, and CFN001 and CFN002 for the client side.

The exception handling tests were directly carried out on the server by invoking the service with abnormal parameters, such as out-of-bound values or invalid arguments (e.g., providing strings where numeric values are expected). It was also useful for testing the behaviour of the service when requests fail to be authenticated. The involved requirements are: SFN007, SFN011, SSC001, SSC002, SSC003, SSC004, SSC005, SSC006, SSC007 and SSC008.

The updater module was tested independently by programming different periodic tasks on the server. This test coressponds to requirement SFN009.

The rest of the tests were executed on the client application as they were directly related to the client itself. These included testing the creation, modification and deletion of several vehicle profiles (requirements CFN008, CFN009, CFN010, CFN011, CFN012, CFN013), the tagging and untagging of several gas stations (CFN007) and the ability of the application to determine the current user's location using different location providers (CFN004, CFN005 and CFN006).

Chapter 6

Conclusions and further development

6.1 Final conclusions

At this point, the EZGas project has become a fully-working client-server application available for any user who owns an Android smartphone. Throughout the process of developing EZGas, the author has carried out different tasks which span several areas of study within the Computer Science degree, such as:

- studying the needs of a fictional "customer" in order to understand what is expected from the application,
- determining the software requirements of the system from the previous study,
- analyzing which resources are already available to achieve the system's goals and which ones need to be created from scratch,
- designing a system architecture which fulfills the specifications and does it in an efficient and effective way,
- creating storage facilities for the application data using different database systems,
- implementing the solution using different programming languages, and
- testing the final product and verifying that it complies with the stated requirements

Ultimately, the goal of this project has been to develop a good product which helps users in an everyday situation such as buying fuel and filling

their vehicles' tanks. It has not been easy, though, due to the time constraints and the continuous development of third-party applications which target the same problem and which implemented new features and bugfixes in every release.

EZGas has been designed to be as simple to use as possible, with clean user interfaces and an intuitive action flow which hopefully will allow its users to become comfortable using it in a short amount of time. The author considers that the final result, albeit not having all the desired functionality, lives to the expectations but also leaves a door open for further improvements and new features.

6.2 Further development

There are some other interesting ideas which can be implemented in an application like EZGas. This section will explain a few concepts which were considered during the initial stages of the project but were later dropped from the final release due to time constraints.

6.2.1 Adding other energy sources

Hybrid and electric cars are becoming popular nowadays due to people shifting to a more ecological and environment-friendly mentality. As of today the number of public locations where users can recharge an electric vehicle is quite small, but that is expected to change within a few years as the technology powering these types of cars becomes cheaper and more robust.

By the time that happens, users should have the chance of discovering where to recharge their cars just the same way EZGas users do today.

6.2.2 Calculation of gas stations en-route

It is not uncommon for drivers to find out their need to refuel while being on the road. In such scenario, which gas station is the absolutely most economical around the user is not as important as diverting from the original route as less as possible.

A way in which this functionality can be implemented is by having users to enter the destination of their trip and calculate a path to it (just as done by any navigation software). The resulting route could then be sampled at fixed distances to obtain reference coordinates, around which the regular searches would be performed within a small area. There is no need to consider the full route, but only those samples which are a few kilometers ahead of the user.

Figure 6.1 illustrates this concept: the light blue line is the route the user is following, the blue dot is the user's current location, the red line is

the look-ahead distance and the green markers are the sampled coordinates with their respective search radii.

Figure 6.1: Calculation of the best en-route gas stations

6.2.3 Providing information about additional services

Most gas stations, usually those found on highways and out of metropolitan areas, not only offer fuel but they also have additional services and facilities such as restaurants or hotels. Currently, the information displayed by EZGas about a gas station relates to the fuel prices, its location and its opening schedule. This information could be expanded to include the additional services offered by gas stations giving the application some added value. Users could also filter results according to whether or not gas stations provide the specific service they are looking for.

6.2.4 Tracking of statistics

Several features could be added regarding the statistics of fuel prices. First, the server could keep a history of fuel prices for each gas station[1]. By doing this, the server would be able to deliver data points for a given fuel type in a given gas station and within a specific period of time, allowing the client application to draw a chart. The server would be able to accept queries of

[1]This would not consume great amounts of disk space as the full list of prices for a single day takes about 1.5 MB, or approximately 550 MB for a full year of data.

up to three or four different values at once, allowing the user to compare prices between different gas stations and their evolution throughout time.

The other planned funcionality was to be implemented directly on the client application: a new section of EZGas would allow users to keep track of every time they refuel their vehicles by entering the date of the operation and the amount of fuel purchased. By collecting this data, users could later review statistics such as the average consumption of their vehicles or the monthly expenses on fuel.

6.2.5 Considering traffic status on calculations

The current implementation of EZGas uses driving distances when calculating distances between two geographic points. However, these distances correspond to the shortest path between them and don't consider other factors which may turn a specific route to be inefficient. One of these factors is the traffic, so EZGas could make use of third-party information services to determine if a specific section of the road network is jammed and, if so, avoid it when calculating the distances.

6.2.6 Augmented reality

As a way of enhancing the user experience, EZGas could implement some augmented reality features such as telling users the exact location of a gas station by using phone components like the camera, the compass, the GPS, the accelerometer or the gyroscope, just like well-known applications like *Layar* already do today.

Appendix A

Project budget

This appendix will provide an estimate of the total project cost broken down into categories such as software, services, human resources and hardware equipment. Please note that the calculated cost belongs to a usage period of six months.

A.1 Hardware equipment

Table A.1 presents the physical equipment used throughout the project. A product life of four years has been considered when calculating costs.

A.2 Software

Several tools have been used during the development of EZGas, as described in section 4.2; however, none impacts the project costs as all of them were available for free.

A.3 Services

Table A.2 shows the third-party services used during the development of the project.

Item	**Price**	**Cost**
Desktop computer (Intel Core 2 Duo, 2.1 GHz)	€700.00	€87.50
D-Link DSL-2640B wireless ADSL router	€90.00	€11.25
Samsung Galaxy S Plus GT-I9001	€250.00	€31.25
Cost of hardware equipment		**€130.00**

Table A.1: Price and cost of hardware equipment

Item	**Price**	**Cost**
ADSL Internet subscription	€30.00	€180.00
Virtual server hosting (512-2048 MB)	€19.99	€119.94
Cost of services		**€299.94**

Table A.2: Price and cost of services

Job position	**Monthly salary**	**Cost**
Software analyst (1 month)	€3,500	€3,500
Software developer (5 months)	€2,500	€12,500
Cost of human resources		**€16,000**

Table A.3: Price and cost of human resources

A.4 Human resources

Table A.3 presents all costs related to the personnel involved in the development of the project.

A.5 Grand total

Table A.4 summarizes the project costs.

Category	**Cost**
Software costs	€0.00
Hardware costs	€130.00
Service costs	€299.94
H&R costs	€16,000
Gross cost	**€16,429.94**
Indirect expenses (15%)	**€2,464.49**
GRAND TOTAL	**€18,894.43**

Table A.4: Total cost of the project

Bibliography

[1] Marc Abrams. *World Wide Web - Beyond the Basics*. Prentice Hall, 2012.

[2] Open Handset Alliance. *Alliance FAQ*. URL: http://www.openhandsetalliance.com/oha_faq.html (visited on 06/10/2012).

[3] *Android Smartphone Activations Reached 331 Million in Q1'2012 Reveals New Device Tracking Database from Signals and Systems Telecom*. May 16, 2012. URL: http://www.prweb.com/releases/2012/5/prweb9514037.htm (visited on 06/10/2012).

[4] *Apple's Phone: From 1980s' Sketches to iPhone. Part 3*. URL: http://mobile-review.com/articles/2010/iphone-history3-en.shtml (visited on 06/10/2012).

[5] Shane Conder and Lauren Darcey. *Android Wireless Application Development*. 2. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN: 978-0-321-74301-5.

[6] Microsoft Corporation. *Microsoft Bing Maps Platform API's Terms of Use*. URL: http://www.microsoft.com/maps/product/terms.html (visited on 06/05/2012).

[7] Oracle Corporation. *About the Java Technology*. URL: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html (visited on 05/12/2012).

[8] Oracle Corporation. *Learn about Java Technology*. URL: http://www.java.com/en/about/ (visited on 05/12/2012).

[9] Oracle Corporation. *MySQL Customers by Industry*. URL: http://www.mysql.com/customers/industry/ (visited on 05/21/2012).

[10] Oracle Corporation. *Sun to Acquire MySQL*. Jan. 16, 2008. URL: http://www.mysql.com/news-and-events/sun-to-acquire-mysql.html (visited on 06/04/2012).

[11] Oracle Corporation. *The Main Features of MySQL*. URL: http://dev.mysql.com/doc/refman/5.1/en/features.html (visited on 05/21/2012).

[12] Jim Davis. *Short Take: BlackBerry wireless email device debuts*. Jan. 20, 1999. URL: http://news.cnet.com/Short-Take-BlackBerry-wireless-email-device-debuts/2110-1040_3-220388.html?tag=mncol (visited on 06/10/2012).

[13] Bruce Eckel. *Thinking in Java (4th Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN: 0131872486.

[14] Evan-Amos. *A BlackBerry Bold 9650 cell phone for the Verizon Wireless phone network*. Aug. 12, 2011. URL: http://commons.wikimedia.org/wiki/File:Blackberry-Bold-9650-Verizon.jpg (visited on 06/10/2012).

[15] Eclipse Foundation. *Juno - Eclipsepedia*. Mar. 17, 2012. URL: http://wiki.eclipse.org/Juno (visited on 06/04/2012).

[16] W. Jason Gilmore. *Beginning PHP and MySQL: From Novice to Professional, Third Edition (Beginning from Novice to Professional)*. 3rd ed. Berkely, CA, USA: Apress, 2008. ISBN: 1590598628, 9781590598627.

[17] Dan Graziano. *Symbian is officially no longer Nokia's problem*. June 23, 2011. URL: http://www.engadget.com/2011/02/11/rip-symbian/ (visited on 06/10/2012).

[18] The PHP Group. *What Is PHP?* May 2012. URL: http://www.php.net/manual/en/intro-whatis.php (visited on 05/12/2012).

[19] Sayed Y. Hashimi et al. *Pro Android 2*. Apress, 2010, pp. 1–24. ISBN: 978-1-4302-2660-4.

[20] Karen Haslam. *Macworld Expo: Optimised OS X sits on 'versatile' flash*. Jan. 12, 2007. URL: http://www.macworld.co.uk/ipad-iphone/news/?newsid=16927 (visited on 06/10/2012).

[21] Sipho Hlongwane. *BBX, BlackBerry's latest purported saviour*. Oct. 21, 2011. URL: http://dailymaverick.co.za/article/2011-10-21-bbx-blackberrys-latest-purported-saviour (visited on 06/10/2012).

[22] Don Ho. *Notepad++ Features*. URL: http://notepad-plus-plus.org/features.html (visited on 06/04/2012).

[23] hostjava.net. *Java Hosting Solutions*. URL: http://www.hostjava.net/ (visited on 06/04/2012).

[24] 1&1 Internet Inc. *Web Hosting*. URL: http://www.1and1.com/Hosting (visited on 06/04/2012).

[25] Google Inc. *Google Maps/Google Earth APIs Tems of Service*. URL: https://developers.google.com/maps/terms (visited on 06/05/2012).

[26] Google Inc. *The Google Directions API*. URL: https://developers.google.com/maps/documentation/directions/ (visited on 06/05/2012).

[27] Google Inc. *What is the NDK? - Android Developers*. URL: http://developer.android.com/sdk/ndk/overview.html (visited on 06/03/2012).

[28] MapQuest Inc. *MapQuest Open Directions API Web Service*. URL: http://developer.mapquest.com/web/products/open/directions-service (visited on 06/05/2012).

[29] J. Liberty and D. Hurwitz. *Programming Asp.net*. O'Reilly Series. O'Reilly, 2005. ISBN: 9780596009168. URL: http://books.google.es/books?id=MTne6h_chwEC.

[30] Ingrid Lunden. *Google Play About To Pass 15 Billion App Downloads? Pssht! It Did That Weeks Ago*. May 7, 2012. URL: http://techcrunch.com/2012/05/07/google-play-about-to-pass-15-billion-downloads-pssht-it-did-that-weeks-ago/ (visited on 06/10/2012).

[31] Mark Lutz. *Learning Python, 3rd edition*. Third. O'Reilly, 2007. ISBN: 9780596513986.

[32] MacRumors. *App Store's 25 Billionth Download Comes From China with 'Where's My Water? Free'*. Mar. 5, 2012. URL: http://www.macrumors.com/2012/03/05/app-stores-25-billionth-download-comes-from-china-with-wheres-my-water-free/ (visited on 06/10/2012).

[33] Nokia. *Nokia 3310 Phone*. URL: http://europe.nokia.com/A4143121 (visited on 06/07/2012).

[34] Nokia. *Nokia 3310 Phone*. URL: http://press.nokia.com/1996/09/19/nokia-unveils-world%C2%92s-first-all-in-one-communicator-for-the-americas/ (visited on 06/08/2012).

[35] Nokia. *Nokia Introduces Nokia 2652, fold design for new growth markets, Major milestone reached - one billionth Nokia mobile phone sold this summer*. Sept. 21, 2005. URL: http://press.nokia.com/2005/09/21/nokia-introduces-nokia-2652-fold-design-for-new-growth-markets-major-milestone-reached-one-billionth-nokia-mobile-phone-sold-this-summer (visited on 06/07/2012).

[36] Chris O'Malley. "Simonizing the PDA". In: *Byte* (1994).

[37] Mike Owens. *The Definitive Guide to SQLite*. Berkeley, CA, USA: Apress, 2006. ISBN: 1590596730.

[38] PDAdb.net. *Nokia 9210 Communicator Specs - Technical Datasheet*. URL: http://pdadb.net/index.php?m=specs&id=886&c=nokia_9210_communicator (visited on 06/08/2012).

[39] Michael Pilato. *Version Control With Subversion*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004. ISBN: 0596004486.

[40] Thomas Ricker. *RIP Symbian*. Feb. 11, 2011. URL: http://www.engadget.com/2011/02/11/rip-symbian/ (visited on 06/10/2012).

[41] Peter Rojas. *It's official: ROKR E1 iTunes phone can only store max. 100 tracks*. Sept. 8, 2005. URL: http://www.engadget.com/2005/09/08/its-official-rokr-e1-itunes-phone-can-only-store-max-100/ (visited on 06/10/2012).

[42] TIOBE Software. *TIOBE Programming Community Index for May 2012*. May 2012. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (visited on 05/12/2012).

[43] StatCounter. *Top 8 Mobile Operating Systems on Apr 2012*. URL: http://gs.statcounter.com/#mobile_os-ww-monthly-201204-201204-bar (visited on 06/10/2012).

[44] *The Apple Of Your Ear*. Jan. 17, 2007. URL: http://www.time.com/time/magazine/article/0,9171,1576854,00.html (visited on 06/10/2012).

[45] Ubuntu. *CronHowTo - Community Ubuntu Documentation*. Sept. 14, 2011. URL: https://help.ubuntu.com/community/CronHowto.

[46] Zach Vega. *An iPhone 4S*. Oct. 12, 2011. URL: http://commons.wikimedia.org/wiki/File:IPhone_4S_No_shadow.png (visited on 06/10/2012).